latools Documentation

Release 0.3.19

Oscar Branson

Apr 06, 2021

Contents

`LAtools`: a Python toolbox for processing Laser Ablations Mass Spectrometry (LA-MS) data.

CHAPTER 1

Overview

1.1 User Guide

1.1.1 Start Here!

If you're completely new to LAtools (and Python!?), these are the steps you need to follow to get going.

1. *Install Python and LAtools*.
2. Go through the *Begginners Guide example data analysis*.
3. *Configure LAtools* for your system.

And you're done!

If you run into problems with the software or documentation, please let us know.

1.1.2 Introduction

Laser Ablation Tools (`latools`) is a Python toolbox for processing Laser Ablations Mass Spectrometry (LA-MS) data.

1.1.2.1 Why Use latools?

At present, most LA-MS data requires a degree of manual processing. This introduces subjectivity in data analysis, and independent expert analysts can obtain significantly different results from the same raw data. At present, there is no standard way of reporting LA-MS data analysis, which would allow an independent user to obtain the same results from the same raw data. `latools` is designed to tackle this problem.

`latools` automatically handles all the routine aspects of LA-MS data reduction:

1. Signal De-spiking
2. Signal / Background Identification

3. Background Subtraction
4. Normalisation to internal standard
5. Calibration to SRMs

These processing steps perform the same basic functions as other LA-MS processing software. If your end goal is calibrated ablation profiles, these can be exported at this stage for external plotting an analysis. The real strength of `latools` comes in the systematic identification and removal of contaminant signals, and calculation of integrated values for ablation spots. This is accomplished with two significant new features.

6. Systematic data selection using quantitative data selection *filters*.
7. Analyses can be fully reproduced by independent users through the export and import of analytical sessions.

These features provide the user with systematic tools to reduce laser ablation profiles to per-ablation integrated averages. At the end of processing, `latools` can export a set of parameters describing your analysis, along with a minimal dataset containing the SRM table and all raw data required to reproduce your analysis (i.e. only analytes explicitly used during processing).

1.1.2.2 Very Important Warning

If used correctly, `latools` will allow the high-throughput, semi-automated processing of LA-MS data in a systematic, reproducible manner. Because it is semi-automated, it is very easy to treat it as a 'black box'. **You must not do this.** The data you get at the end will only be valid if processed *appropriately*. Because `latools` brings reproducibility to LA-MS processing, it will be very easy for peers to examine your data processing methods, and identify any shortfalls. In essence: to appropriately use `latools`, you must understand how it works!

The best way to understand how it works will be to play around with data processing, but before you do that there are a few things you can do to start you off in the right direction:

1. Read and understand the following 'Overview' section. This will give you a basic understanding of the architecture of `latools`, and how its various components relate to each other.
2. Work through the 'Getting Started' guide. This takes you step-by-step through the analysis of an example dataset.
3. Be aware of the extensive documentation that describes the action of each function within `latools`, and tells you what each of the input parameters does.

1.1.2.3 Overview: Understand `latools`

`latools` is a Python 'module'. You do not need to be fluent in Python to understand `latools`, as understanding *what* each processing step does to your data is more important than *how* it is done. That said, an understanding of Python won't hurt!

Architecture

The `latools` module contains two core 'objects' that interact to process LA-MS data:

- `latools.D` is the most 'basic' object, and is a 'container' for the data imported from a single LA-MS data file.
- `latools.analyse` is a higher-level object, containing numerous `latools.D` objects. This is the object you will interact with most when processing data, and it contains all the functions you need to perform your analysis.

This structure reflects the hierarchical nature of LA-MS analysis. Each ablation contains an measurements of a single sample (i.e. the 'D' object), but data reduction requires consideration of multiple ablations of samples and standards collected over an analytical session (i.e. the 'analyse' object). In line with this, some data processing steps (de-spiking, signal/background identification, normalisation to internal standard) can happen at the individual analysis level (i.e. within the `latools.D` object), while others (background subtraction, calibration, filtering) require a more holistic approach that considers the entire analytical session (i.e. at the `latools.analyse` level).

How it works

In practice, you will do all data processing using the `latools.analyse` object, which contains all the data processing functionality you'll need. To start processing data, you create an `latools.analyse` object and tell it which folder your data are stored in. `latools.analyse` then imports all the files in the data folder as `latools.D` objects, and labels them by their file names. The `latools.analyse` object contains all of the `latools.D` objects withing a 'dictionary' called `latools.analyse.data_dict`, where the each individual `latools.D` object can be accessed via its name. Data processing therefore works best when ablations of each individual sample or standard are stored in a single data folder, named according to what was measured.

Todo: In the near future, `latools` will also be able to cope with multiple ablations stored in a single, long data file, as long as a list of sample names is provided to identify each ablation.

When you're performing a processing step that can happen at an individual-sample level (e.g. de-spiking), the `latools.analyse` object passes the task directly on to the `latools.D` objects, whereas when you're performing a step that requires consideration of the *entire* analytical session (e.g. calibration), the `latools.analyse` object will coordinate the interaction of the different `latools.D` objects (i.e. calculate calibration curves from SRM measurements, and apply them to quantify the compositions of your unknown samples).

Filtering

Finally, there is an additional 'object' attached to each `latools.D` object, specifically for handling data filtering. This `latools.filt` object contains all the information about filters that have been calculated for the data, and allows you to switch filters on or off for individual samples, or subsets of samples. This is best demonstrated by example, so we'll return to this in more detail in the *Data Selection and Filtering* section of the *Beginner's Guide*

1.1.2.4 Where next?

Hopefully, you now have a rudimentary understanding of how `latools` works, and how it's put together. To start using `latools`, *install* it on your system, then work through the step-by-step example in the *Beginner's Guide* guide to begin getting to grips with how `latools` works. If you already know what you're doing and are looking for more in-depth information, head to advanced_topics, or use the search bar in the top left to find specific information.

1.1.3 Installation

1.1.3.1 Prerequisite: Python

Before you install `latools`, you'll need to make sure you have a working installation of Python, **preferably Python 3**. If you don't already have this (or are unsure if you do), we recommend that you install one of the pre-packaged science-oriented Python distributions, like Continuum's Anaconda. This provides a working copy of Python, and most of the modules that `latools` relies on.

If you already have a working Python installation or don't want to install one of the pre-packaged Python distributions, everything below *should* work.

Tip: Make sure you set the Anaconda Python installation as the system default, or you are working in a virtual environment that uses the correct Python version. If you don't know what a virtual environment' is, don't worry - just make sure you check the box saying 'make this my default Python' at the appropriate time when installing Anaconda.

1.1.3.2 Installing `latools`

There are two ways to install `latools`. We recommend the first method, which will allow you to easily keep your installation of `latools` up to date with new developments.

Both methods require entering commands in a terminal window. On Mac, open the **Terminal** application in '/Applications/Utilities'. On Windows, this is a little more complex - see instructions here.

1. Using `pip`

```
pip install latools
```

For in-depth instructions on using *pip*, see here

2. Using `conda`

Coming soon...

1.1.3.3 Next Steps

If this is your first time, read through the *Getting Started* guide. Otherwise, get analysing!

1.1.4 Beginner's Guide

1.1.4.1 Getting Started

This guide will take you through the analysis of some example data included with `latools`, with explanatory notes telling you what the software is doing at each step. We recommend working through these examples to understand the mechanics of the software before setting up your *Three Steps to Configuration*, and working on your own data.

The Fundamentals: Python

Python is an open-source (free) general purpose programming language, with growing application in science. `latools` is a python *module* - a package of code containing a number of Python *objects* and functions, which run within Python. That means that you need to have a working copy of Python to use `latools`.

If you don't already have this (or are unsure if you do), we recommend that you install one of the pre-packaged science-oriented Python distributions, like Continuum's Anaconda (recommended). This provides a complete working installation of Python, and all the pre-requisites you need to run latools.

latools has been developed and tested in Python 3.5. It *should* also run on 2.7, but we can't guarantee that it will behave.

''latools" should work in any python interpreter, but we recommend either Jupyter Notebook or iPython. Jupyter is a browser-based interface for ipython, which provides a nice clean interactive front-end for writing code, taking notes and viewing plots.

For simplicity, the rest of this guide will assume you're using Jupyter notebook, although it should translate directly to other Python interpreters.

For a full walk through of getting latools set up on your system, head on over to the *Installation* guide.

Preparation

Before we start latools, you should create a folder to contain everything we're going to do in this guide. For example, you might create a folder called latools_demo/ on your Desktop - we'll refer to this folder as latools_demo/ from now on, but you can call it whatever you want. Remember where this folder is - we'll come back to it a lot later.

Tip: As you process data with latools, new folders will be created in this directory containing plots and exported data. This works best (i.e. is least cluttered) if you put your data in a single directory inside a parent directory (in this case latools_demo), so all the directories created during analysis will also be in the same place, without lots of other files.

Starting *latools*

Next, launch a Jupyter notebook in this folder. To do this, open a terminal window, and run:

```
cd ~/path/to/latools_demo/
jupyter notebook
```

This should open a browser window, showing the Jupyter main screen. From here, start a new Python notebook by clicking 'New' in the top right, and selecting your Python version (preferably 3.5+). This will open a new browser tab containing your Jupyter notebook.

Once python is running, import latools into your environment:

```
import latools as la
```

All the functions of latools are now accessible from within the la prefix.

Tip: if you want Jupyter notebook to display plots in-line (recommended), add an additional line after the import statement: %matplotlib inline.

Tip: To run code in a Jupyter notebook, you must 'evaluate' the cell containing the code. to do this, type:

- [ctrl] + [return] evaluate the selected cell.
- [shift] + [return] evaluate the selected cell, and moves the focus to the next cell
- [alt] + [return] evaluate the selected cell, and creates a new empty cell underneath.

Example Data

Once you've imported `latools`, extract the example dataset to a `data/` folder within `latools_demo/`:

```
la.get_example_data('./latools_demo_tmp')
```

Take a look at the contents of the directory. You should see four .csv files, which are raw data files from an Agilent 7700 Quadrupole mass spectrometer, outputting the counts per second of each analyte as a function of time. Notice that each .csv file either has a sample name, or is called 'STD'.

Note: Each data file should contain data from a single sample, and data files containing measurements of standards should all contain an identifying set of characters (in this case 'STD') in the name. For more information, see *Data Formats*.

1.1.4.2 Importing Data

Once you have Python running in your `latools_demo/` directory and have unpacked the *Example Data*, you're ready to start an `latools` analysis session. To do this, run:

```
eg = la.analyse(data_folder='./latools_demo_tmp',  # the location of your data
                config='UCD-AGILENT',  # the configuration to use
                internal_standard='Ca43',  # the internal standard in your analyses
                srm_identifier='STD')  # the text that identifies which files contain␣
↪standard reference materials
```

This imports all the data files within the `data/` folder into an `latools.analyse` object called `eg`, along with several parameters describing the dataset and how it should be imported:

- `config='DEFAULT'`: The configuration contains information about the data file format and the location of the SRM table. Multiple configurations can be set up and chosen during data import, allowing `latools` to flexibly work with data from different instruments.
- `internal_standard='Ca43'`: This specifies the internal standard element within your samples. The internal standard is used at several key stages in analysis (signal/background identification, normalisation), and should be relatively abundant and homogeneous in your samples.
- `srm_identifier='STD'`: This identifies which of your analyses contain standard reference materials (SRMs). Any data file with 'STD' in its name will be flagged as an SRM measurement.

Tip: You've just created an analysis called *eg*. Everything we'll do from this point on happens within that analysis session, so you'll see *eg.some_function()* a lot. When doing this yourself, you can give your analysis any name you want - you *don't* have to call it *eg*, but if you change the name of your analysis to *my_analysis* remember that *eg.some_funtion()* will no longer work - you'll have to use *my_analysis.some_function()*.

If it has worked correctly, you should see the output:

```
latools analysis using "DEFAULT" configuration:
  5 Data Files Loaded: 2 standards, 3 samples
  Analytes: Mg24 Mg25 Al27 Ca43 Ca44 Mn55 Sr88 Ba137 Ba138
  Internal Standard: Ca43
```

In this output, `latools` reports that 5 data files were imported from the `data/` directory, two of which were standards (names contained 'STD'), and tells you which analytes are present in these data. Each of the imported data files is stored in a `latools.D` object, which are 'managed' by the `latools.analyse` object that contains them.

Tip: `latoools` expects data to be organised in a *particular way*. If your data do not meet these specifications, have a read through the *Pre-Processing Guide* for advice on getting your data in the right format.

Check inside the `latools_demo` directory. There should now be two new folders called `reports_data/` and `export_data/` alongside the `data/` folder. Note that the '_data' suffix will be the same as the name of the folder that contains your data - i.e. the names of these folders will change, depending on the name of your data folder. `latools` saves data and plots to these folders throughout analysis:

- `data_export` will contain exported data: traces, per-ablation averages and minimal analysis exports.
- `data_reports` will contain all plots generated during analysis.

1.1.4.3 Plotting

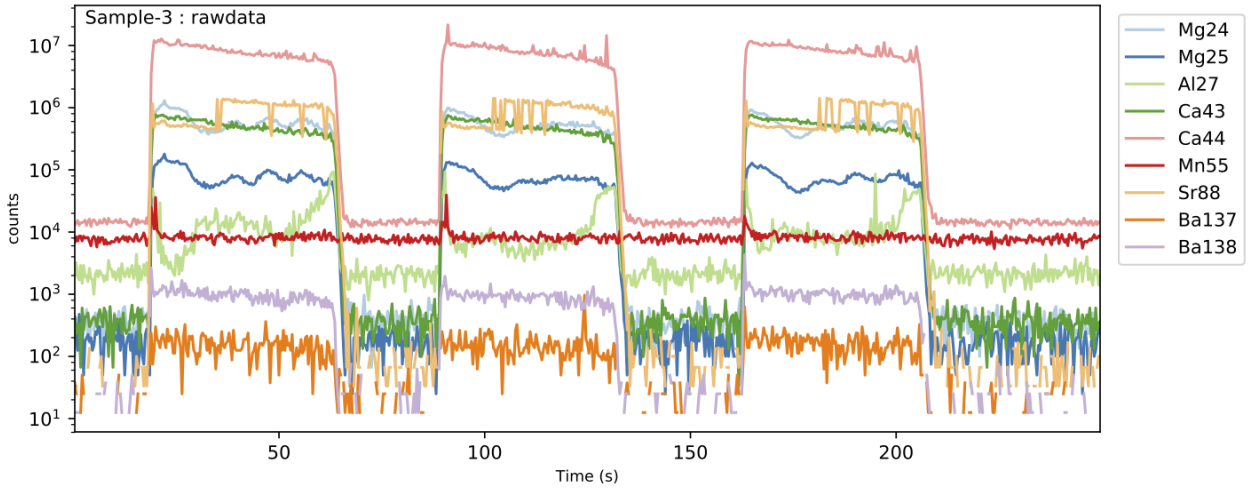
Danger: Because `latools` offers the possibility of high-throughput analyses, it will be tempting to use it as an analytical 'black box'. **DO NOT DO THIS**. It is *vital* to keep track of your data, and make sure you understand the processing being applied to it. The best way of doing this is by *looking* at your data.

The main way to do this in `latools` is to **Plot** all your data, or subsets of samples and analytes, at any stage of analysis using `trace_plots()`. The resulting plots are saved as pdfs in the `reports_data` folder created during import, in a subdirectory labelled as the analysis stage. For example, making plots now will create 5 plots in a subdirectory called `rawdata`:

```
eg.trace_plots()
```

Tip: Plot appearance can be modified by specifying a range of parameters in this function. This will be used to some extent later in this tutorial, but see `trace_plots()` documentation for more details.

By default all analytes from the most recent stage of analysis are plotted on a log scale, and the plot should look something like this:



Once you've had a look at your data, you're ready to start processing it.

1.1.4.4 Data De-spiking

The first step in data reduction is the 'de-spike' the raw data to remove physically unrealistic outliers from the data (i.e. higher than is physically possible based on your system setup).

Two de-spiking methods are available:

- `expdecay_despiker()` removes low outliers, based on the signal washout time of your laser cell. The signal washout is described using an exponential decay function. If the measured signal decreases faster than physically possible based on your laser setup, these points are removed, and replaced with the average of the adjacent values.
- `noise_despiker()` removes high outliers by calculating a rolling mean and standard deviation, and replacing points that are greater than *n* standard deviations above the mean with the mean of the adjacent data points.

These functions can both be applied at once, using `despike()`:

```
eg.despike(expdecay_despiker=True,
           noise_despiker=True)
```

By default, this applies `expdecay_despiker()` followed by `noise_despiker()` to all samples. You can specify several parameters that change the behaviour of these de-spiking routines.

The `expdecay_despiker()` relies on knowing the exponential decay constant that describes the washout characteristics of your laser ablation cell. If this values is missing (as here), `latools` calculates it by fitting an exponential decay function to the internal standard at the on-off laser transition at the end of ablations of standards. If this has been done, you will be informed. In this case, it should look like:

```
Calculating exponential decay coefficient
from SRM Ca43 washouts...
  -2.28
```

Tip: The exponential decay constant used by `expdecay_despiker()` will be specific to your laser setup. If you don't know what this is, `despike()` determines it automatically by fitting an exponential decay function to the washout phase of measured SRMs in your data. You can look at this fit by passing `exponent_plot=True` to the function.

1.1.4.5 Background Correction

The de-spiked data must now be background-corrected. This involves three steps:

1. Signal and background identification.
2. Background calculation underlying the signal regions.
3. Background subtraction from the signal.

Signal / Background Separation

This is achieved automatically using `autorange()` using the internal standard (Ca43, in this case), to discriminate between 'laser off' and 'laser on' regions of the data. Fundamentally, 'laser on' regions will contain high counts, while 'laser off' will contain low counts of the internal standard. The mid point between this high and low offers a good starting point to approximately identify 'signal' and 'background' regions. Regions in the ablation with higher counts than the mid point are labelled 'signal', and lower are labelled 'background'. However, because the transition between

laser-on and laser-off is not instantaneous, both signal and background identified by this mid-point will contain part of the 'transition', which must be excluded from both signal and background. This is accomplished by a simple algorithm, which determines the width of the transition and excludes it:

1. Extract each approximate transition, and calculate the first derivative. As the transition is approximately sigmoid, the first derivative is approximately Gaussian.
2. Fit a Gaussian function to the first derivative to determine its width. This fit is weighted by the distance from the initial transition guess.
3. Exclude regions either side of the transitions from both signal and background regions, based on the full-width-at-half-maximum (FWHM) of the Gaussian fit. The pre- and post-transition exclusion widths can be specified independently for 'off-on' and 'on-off' transitions.

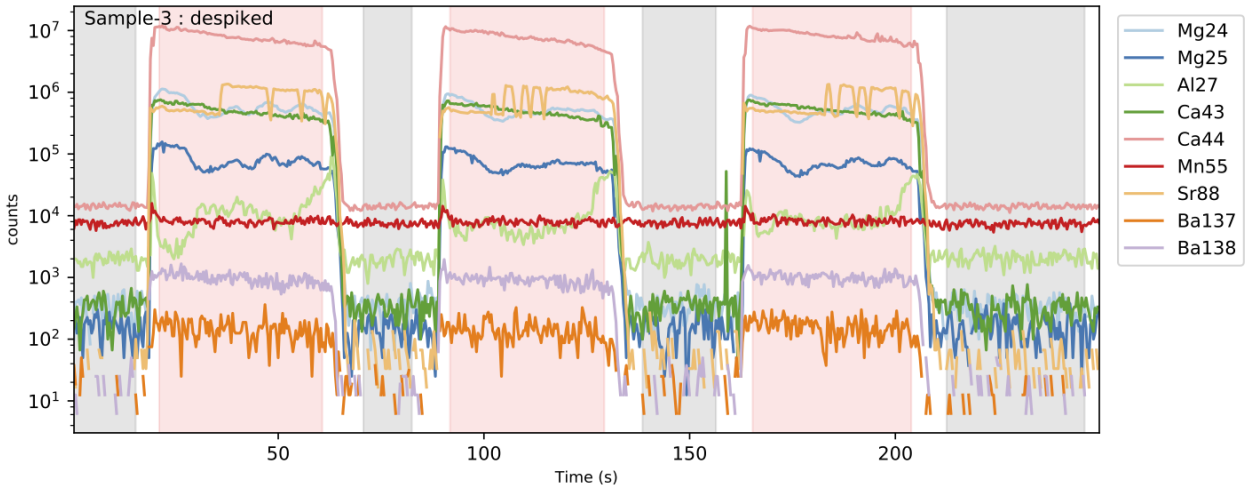
Several parameters within `autorange()` can be modified to subtly alter the behaviour of this function. However, in testing the automatic separation proved remarkably robust, and you should not have to change these parameters much.

The function is applied to your data by running:

```
eg.autorange(on_mult=[1.5, 0.8],
             off_mult=[0.8, 1.5])
```

In this case, `on_mult=[1.5, 0.8]` signifies that a 1.5 x FWHM of the transition will be removed *before* the off-on transition (on the 'background' side), and 0.8 x FWHM will be removed *after* the transition (on the 'signal' side), and vice versa for the on-off transition. This excludes more from the background than the signal, avoiding spuriously high background values caused by the tails of the signal region.

Tip: Look at your data! You can see the regions identified as 'signal' and 'background' by this algorithm by plotting your data using `eg.trace_plots(ranges=True)`. Because the analysis has progressed since the last time you plotted (the data have been de-spiked), these plots will be saved in a new `de-spiked` sub-folder within the `reports_data` folder. This will produce plots with 'signal' regions highlighted in red, and 'background' highlighted in grey:



Background Calculation

Once the background regions of the ablation data have been identified, the background underlying the signal regions must be calculated. At present, `latools` includes two background calculation algorithms:

- `bkg_calc_interp1d()` fits a polynomial function to all background regions, and calculates the intervening background values using a 1D interpolation (numpy's interp1D function). The order of the polynomial can be specified by the 'kind' variable, where kind=0 simply interpolates the mean background forward until the next measured background region.
- `bkg_calc_weightedmean()` calculates a Gaussian-weighted moving average, such that the interpolated background at any given point is determined by adjacent background counts on either side of it, with the closer (in Time) being proportionally more important. The full-width-at-half-maximum (FWHM) of the Gaussian weights must be specified, and should be greater than the time interval between background measurements, and less than the time-scale of background drift expected on your instrument.

Warning: Use extreme caution with polynomial backgrounds of order>1. You should only use this if you know you have significant non-linear drift in your background, which you understand but cannot be dealt with by changing you analytical procedure. In all tested cases the weighted mean background outperformed the polynomial background calculation method.

Note: Other background fitting functions can be easily incorporated. If you're Python-literate, we welcome your contributions. If not, get in touch!

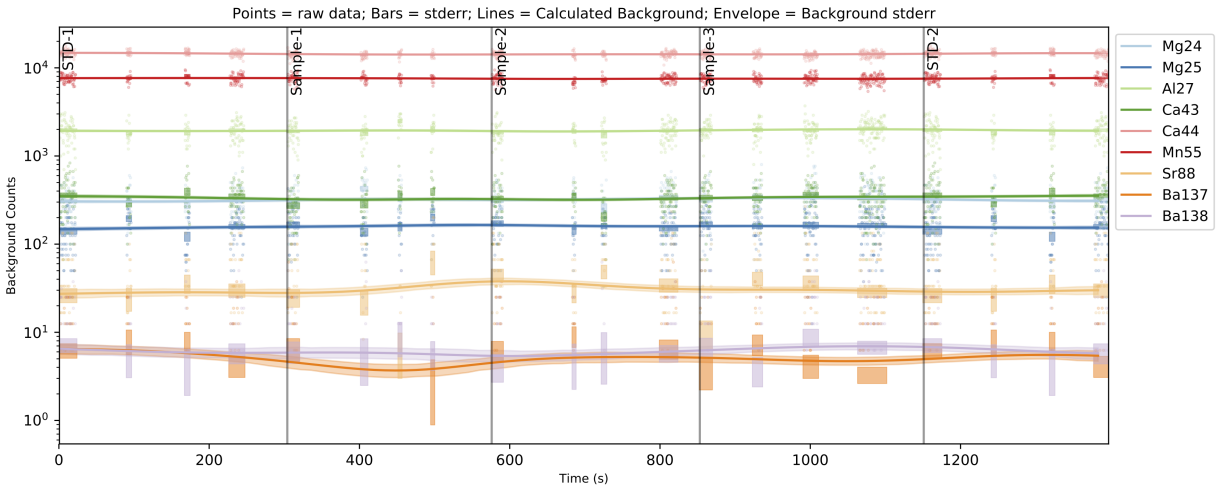
For this demonstration, we will use the `bkg_calc_weightedmean()` background, with a FWHM of 5 minutes (`weight_fwhm=300` seconds), that only considers background regions that contain greater than 10 points (`n_min=10`):

```
eg.bkg_calc_weightedmean(weight_fwhm=300,
                         n_min=10)
```

and plot the resulting background:

```
eg.bkg_plot()
```

which is saved in the `reports_data` subdirectory, and should look like this:



Background Subtraction

Once the background is calculated, it subtracted from the signal regions using `bkg_correct()`:

```
eg.bkg_subtract()
```

Tip: Remember that you can plot the data and examine it at any stage of your processing. running `eg.trace_plots()` now would create a new subdirectory called 'bkgcorrect' in your 'reports_data' directory, and plot all the background corrected data.

1.1.4.6 Ratio Calculation

Next, you must normalise your data to an internal standard, using `ratio()`:

```
eg.ratio()
```

The internal standard is specified during data import, but can also be changed here by specifying `internal_standard` in `ratio()`. In this case, the internal standard is Ca43, so all analytes are divided by Ca43.

1.1.4.7 Calibration

Once all your data are normalised to an internal standard, you're ready to calibrate the data. This is done by creating a calibration curve for each element based on SRMs measured throughout your analysis session, and a table of known SRM values. You can either calculate a single calibration from a combination of all your measured standards, or generate a time-sensitive calibration to account for sensitivity drift through an analytical session. The latter is achieved by creating a separate calibration curve for each element in each SRM measurement, and linearly extrapolating these calibrations between neighbouring standards.

Calibration is performed using the `calibrate()` method:

```
eg.calibrate(drift_correct=False,
             srms_used=['NIST610', 'NIST612', 'NIST614'])
```

In this simple example case, our analytical session is very short, so we are not worried about sensitivity drift (`drift_correct=False`).

There is also a default parameter `poly_n=0`, which specifies that the polynomial calibration line fitted to the data that is forced through zero. Changing this number alters the order of polynomial used during calibration. Because of the wide-scale linearity of ICPM-MS detectors, `poly_n=0` should normally provide an adequate calibration line. If it does not, it suggests that either one of your 'known' SRM values may be incorrect, or there is some analytical problem that needs to be investigated (e.g. interferences from other elements). Finally, `srms_used` contains the names of the SRMs measured throughout analysis. The SRM names you give must *exactly* (case sensitive) match the SRM names in the SRM table.

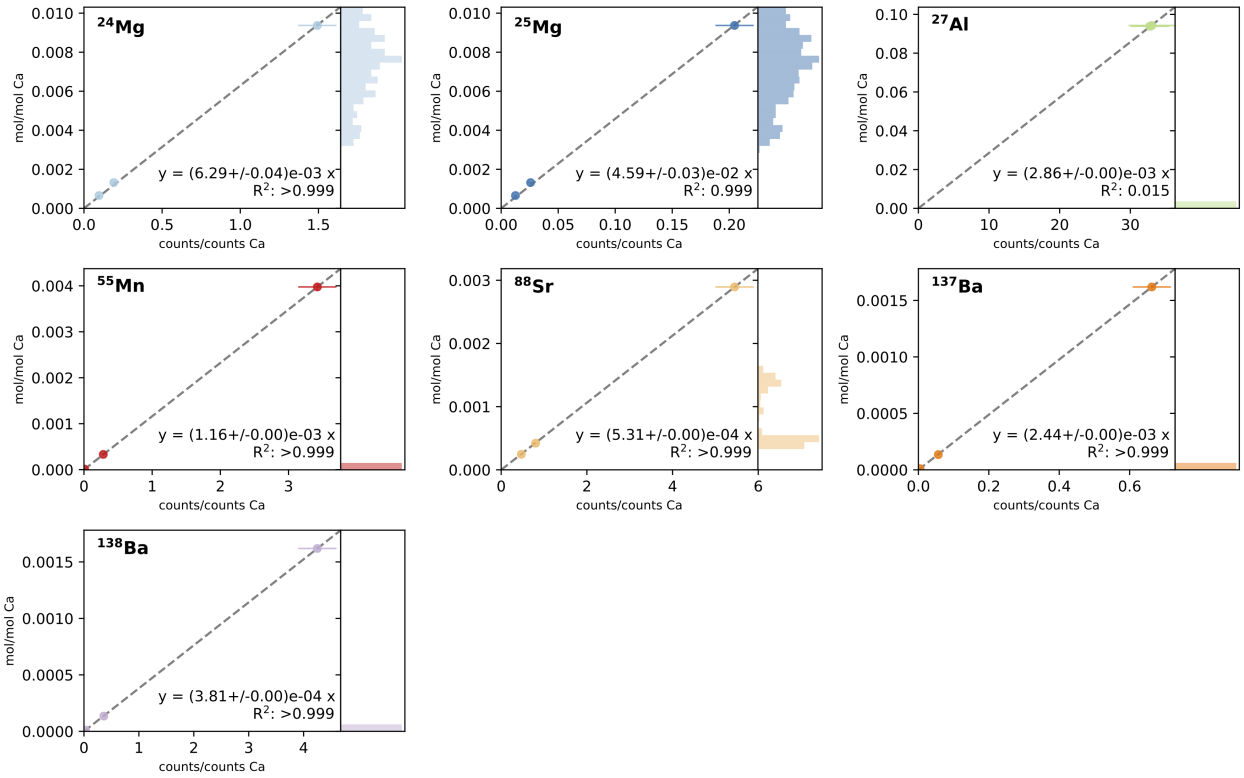
Note: For calibration to work, you must have an SRM table containing the element/internal_standard ratios of the standards you've measured, whose location is specified in the latools configuration. You should only need to do this once for your lab, but it's important to ensure that this is done correctly. For more information, see the *Three Steps to Configuration* section.

First, `latools` will automatically determine the identity of measured SRMs throughout your analysis session using a relative concentration matrix (see SRM Identification section for details). Once you have identified the SRMs in your standards, `latools` will import your SRM data table (defined in the configuration file), calculate a calibration curve for each analyte based on your measured and known SRM values, and apply the calibration to all samples.

The calibration lines for each analyte can be plotted using:

```
eg.calibration_plot()
```

Which should look something like this:



Where each panel shows the measured counts/count (x axis) vs. known mol/mol (y axis) for each analyte with associated errors, with the fitted calibration line, equation and R2 of the fit. The axis on the right of each panel contains a histogram of the raw data from each sample, showing where your sample measurements lie compared to the range of the standards.

1.1.4.8 Mass Fraction (ppm) Calculation

After calibration, all data are in units of mol/mol. For many use cases (e.g. carbonate trace elements) this will be sufficient, and you can continue on to *Data Selection and Filtering*. In other cases, you might prefer to work mass fractions (e.g. ppm). If so, the next step is to convert your mol/mol ratios to mass fractions.

This requires knowledge of the concentration of the internal standard in all your samples, which we must provide to latools. First, generate a list of samples in a spreadsheet:

```
eg.get_sample_list()
```

This will create a file containing a list of all samples in your analysis, with an empty column to provide the mass fraction (or % or ppm) of the internal standard for each individual sample. Enter this information for each sample, and save the file without changing its format (.csv) - remember where you saved it, you'll use it in the next step! Pay

attention to units here - the calculated mass fraction values for your samples will have the same units as you provide here.

Tip: If all your samples have the same concentration of internal standard, you can skip this step and just enter a single mass fraction value at the calculation stage.

Next, import this information and use it to calculate the mass fraction of each element in each sample:

```
eg.calculate_mass_fraction('/path/to/internal_standard_massfrac.csv')
```

Replace *path/to/interninternal_standard_massfrac.csv* with the location of the file you edited in the previous step). This will calculate the mass fractions of all analytes in all samples in the same units as the provided internal standard concentrations. If you know that all your samples have the same internal standard concentration, you could just provide a number instead of a file path here.

1.1.4.9 Data Selection and Filtering

The data are now background corrected, normalised to an internal standard, and calibrated. Now we can get into some of the new features of `latools`, and start thinking about **data filtering**.

This section will tell you the basics - what a filter is, and how to create and apply one. For most information on the different types of filters available in `latools`, head over to the *Filters* section.

What is Data Filtering?

Laser ablation data are spatially resolved. In heterogeneous samples, this means that the concentrations of different analytes will change within a single analysis. This compositional heterogeneity can either be natural and expected (e.g. Mg/Ca variability in foraminifera), or caused by compositionally distinct contaminant phases included in the sample structure. If the end goal of your analysis is to get integrated compositional estimates for each ablation analysis, how you deal with sample heterogeneity becomes central to data processing, and can have a profound effect on the resulting integrated values. So far, heterogeneous samples tend to be processed manually, by choosing regions to integrate by eye, based on a set of criteria and knowledge of the sample material. While this is a valid approach to data reduction, it is not reproducible: if two 'expert analysts' were to process the data, the resulting values would not be quantitatively identical. Reproducibility is fundamental to sound science, and the inability to reproduce integrated values from identical raw data is a fundamental flaw in Laser Ablation studies. In short, this is a serious problem.

To get round this, we have developed 'Data Filters'. Data Filters are systematic selection criteria, which can be applied to all samples to select specific regions of ablation data for integration. For example, the analyst might apply a filter that removes all regions where a particular analyte exceeds a threshold concentration, or exclude regions where two contaminant elements co-vary through the ablation. Ultimately, the choice of selection criteria remains entirely subjective, but because these criteria are quantitative they can be uniformly applied to all specimens, and most importantly, reported and reproduced by an independent researcher. This removes significant possibilities for 'human error' from data analysis, and solves the long-standing problem of reproducibility in LA-MS data processing.

Data Filters

`latools` includes several filtering functions, which can be created, combined and applied in any order, repetitively and in any sequence. By their combined application, it should be possible to isolate any specific region within the data that is systematically identified by patterns in the ablation profile. These filter are (in order of increasing complexity):

- `filter_threshold()`: Creates two filter keys identifying where a specific analyte is above or below a given threshold.

- `filter_distribution()`: Finds separate *populations* within the measured concentration of a single analyte within by creating a Probability Distribution Function (PDF) of the analyte within each sample. Local minima in the PDF identify the boundaries between distinct concentrations of that analyte within your sample.
- `filter_clustering()`: A more sophisticated version of `filter_distribution()`, which uses data clustering algorithms from the sklearn module to identify compositionally distinct 'populations' in your data. This can consider multiple analytes at once, allowing for the robust detection of distinct compositional zones in your data using n-dimensional clustering algorithms.
- `filter_correlation()`: Finds regions in your data where two analytes correlate locally. For example, if your analyte of interest strongly co-varies with an analyte that is a known contaminant indicator, the signal is likely contaminated, and should be discarded.

It is also possible to 'train' a clustering algorithm based on analyte concentrations from *all* samples, and then apply it to individual filters. To do this, use:

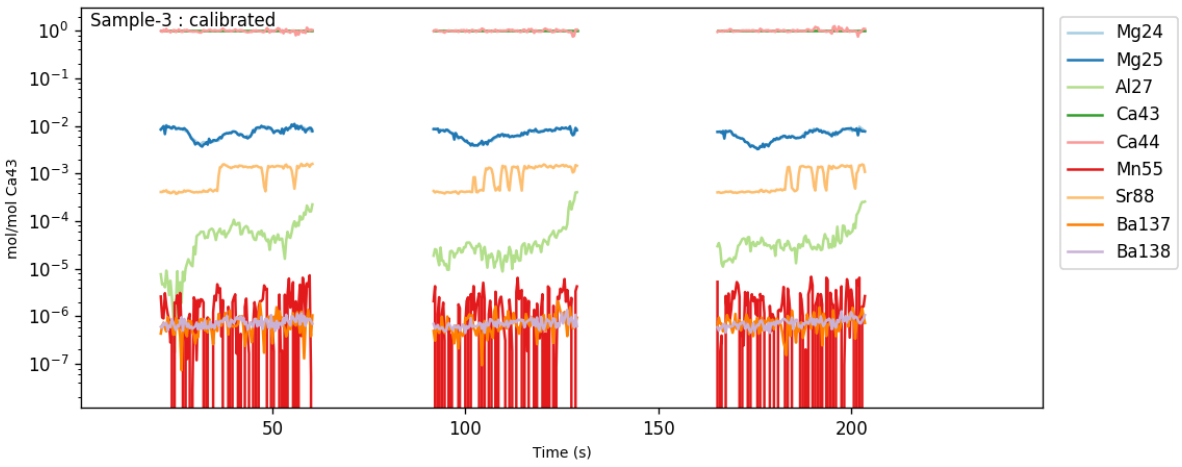
- `fit_classifier()`: Uses a clustering algorithm based on specified analytes in *all* samples (or a subset) to identify separate compositions within the entire dataset. This is particularly useful if (for example) all samples are affected by a contaminant with a unique composition, or the samples contain a chemical 'label' that identifies a particular material. This will be most robustly identified at the whole-analysis level, rather than the individual-sample level.
- `apply_classifier()`: Applies the classifier fitted to the entire dataset to all samples individually. Creates a sample-level filter using the classifier based on all data.

For a full account of these filters, how they work and how they can be used, see *Filters*.

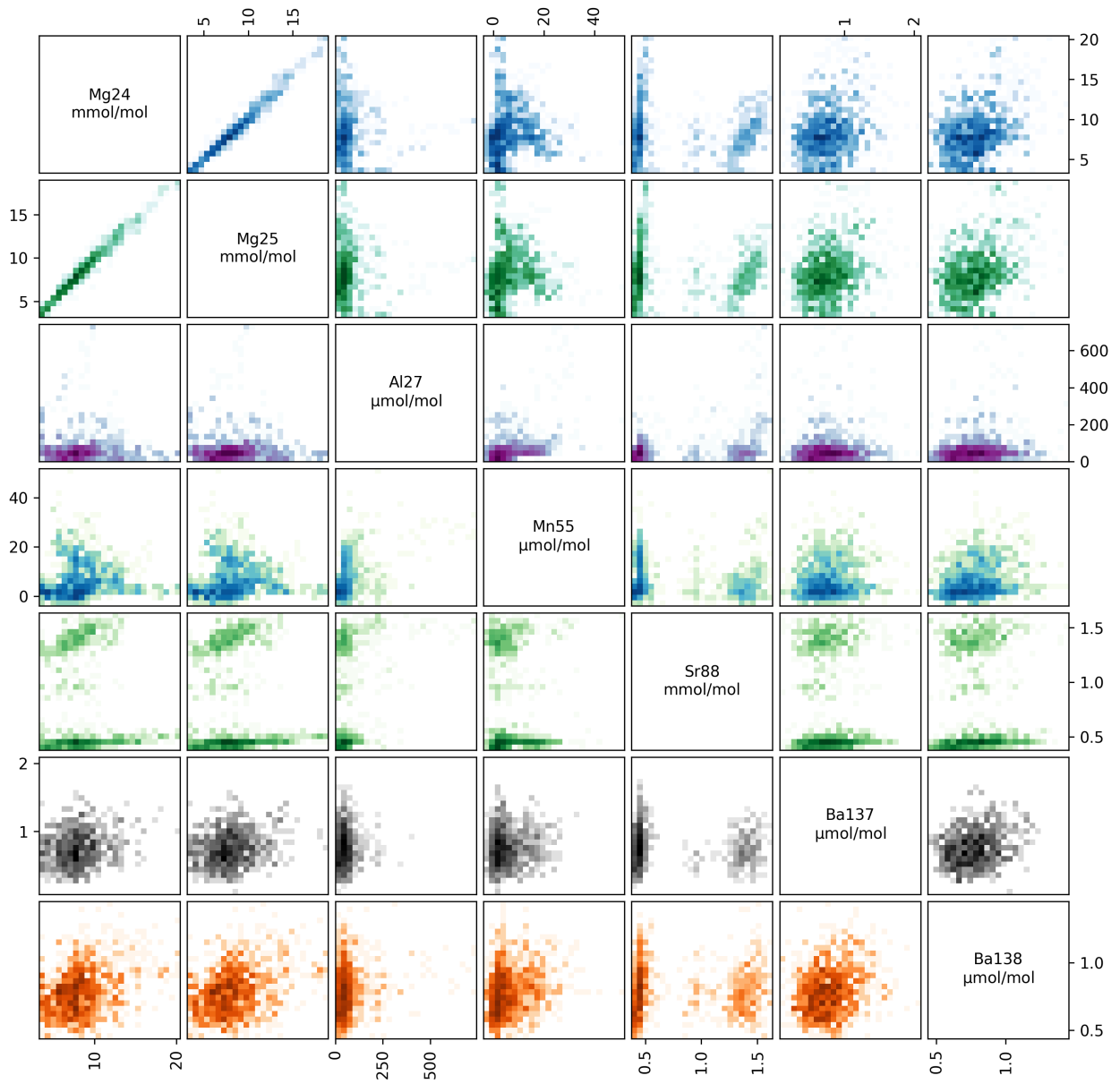
Simple Demonstration

Choosing a filter

The foraminifera analysed in this example dataset are from culture experiments and have been thoroughly cleaned. There should not be any contaminants in these samples, and filtering is relatively straightforward. The first step in choosing a filter is to *look* at the data. You can look at the calibrated profiles manually to get a sense of the patterns in the data (using `eg.trace_plots()`):



Or alternatively, you can make a 'crossplot' (using `eg.crossplot()`) of your data, to examine how all the trace elements in your samples relate to each other:



This plots every analyte in your ablation profiles, plotted against every other analyte. The axes in each panel are described by the diagonal analyte names. The colour intensity in each panel corresponds to the data density (i.e. it's a 2D histogram!).

Within these plots, you should focus on the behaviour of 'contaminant indicator' elements, i.e. elements that are normally within a known concentration range, or are known to be associated with a possible contaminant phase. As these are foraminifera, we will pay particularly close attention to the concentrations of Al, Mn and Ba in the ablations, which are all normally low and homogeneous in foraminifera samples, but are prone to contamination by clay particles. In these samples, the Ba and Mn are relatively uniform, but the Al increases towards the end of each ablation. This is because the tape that the specimens were mounted on contains a significant amount of Al, which is picked up by the laser as it ablates through the shell. We know from experience that the tape tends to have very low concentration of other elements, but to be safe we should exclude regions with hi Al/Ca from our analysis.

Creating a Filter

We wouldn't expect cultured foraminifera to have a Al/Ca of ~100 µmol/mol, so we therefore want to remove all data from regions with an Al/Ca above this. We'll do this with a threshold filter:

```
eg.filter_threshold(analyte='Al27', threshold=100e-6)  # remember that all units are␣
↪in mol/mol!
```

This goes through *all* the samples in our analysis, and works out which analyses have an Al/Ca both greater than and less than 100 µmol/mol (remember, all units are in mol/mol at this stage). This function calculates the filters, but does not apply them - that happens later. Once the filters are calculated, a list of filters and their current status is printed:

```
Subset All_Samples:
Samples: Sample-1, Sample-2, Sample-3

n  Filter Name          Mg24   Mg25   Al27   Ca43   Ca44   Mn55   Sr88   Ba137  Ba138
0  Al27_thresh_below    False  False  False  False  False  False  False  False  False
1  Al27_thresh_above    False  False  False  False  False  False  False  False  False
```

You can also check this manually at any time using:

```
eg.filter_status()
```

This produces a grid showing the filter numbers, names, and which analytes they are active for (for each analyte False = inactive, True = active). The `filter_threshold` function has generated two filters: one identifying data above the threshold, and the other below it. Finally, notice also that it says 'Subset: All_Samples' at the top, and lists which samples they are. You can apply different filters to different subsets of samples... We'll come back to this later. This display shows all the filters you've calculated, and which analytes they are applied to.

Before we think about applying the filter, we should check what it has actually done to the data.

Note: Filters do not delete any data. They simply create a *mask* which tells latools functions which data to use, and which to ignore.

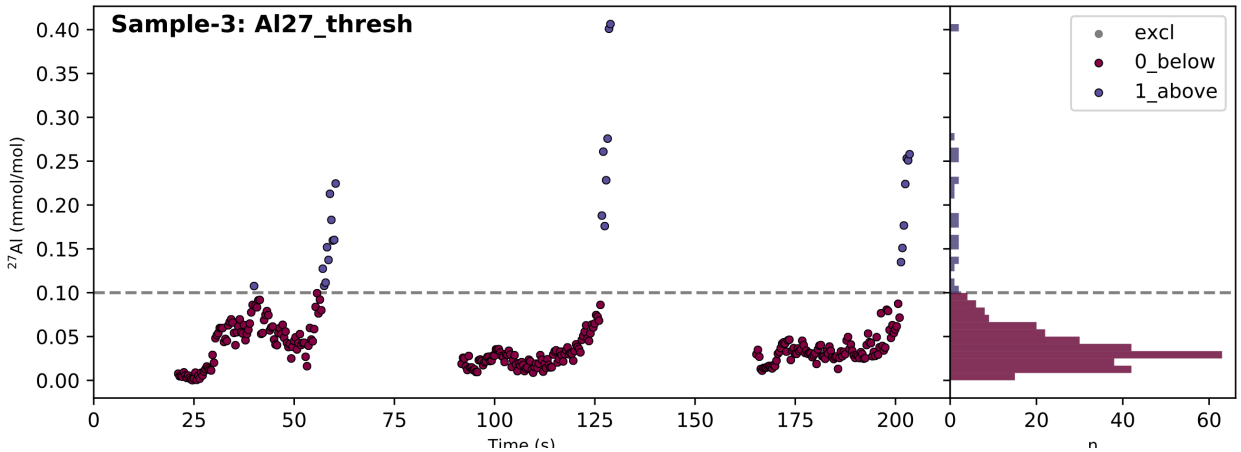
Checking a Filter

You can do this in three ways:

1. Plot the traces, with `filt=True`. This plots the calibrated traces, with areas excluded by the filter shaded out in grey. Specifying `filt=True` shows the net effect of all active filters. By setting `filt` as a number or filter name, the effect of one individual filter will be shown.
2. Crossplot with `filt=True` will generate a new crossplot containing only data that remains after filtering. This can be useful for refining filter choices during multiple rounds of filtering. You can also set `filt` to be a filter name or a number, as with trace plotting.
3. The most sophisticated way of looking at a filter is by creating a 'filter_report'. This generates a plot of each analysis, showing which regions are selected by particular filters:

```
eg.filter_reports(analytes='Al27', filt_str='thresh')
```

Where `analytes` specifies which analytes you want to see the influence of the filters on, and `filt_str` identifies which filters you want to see. `filt_str` supports partial filter name matching, so 'thresh' will pick up any filter with 'thresh' in the name - i.e. if you'd calculated multiple thresholds, it would plot each on a different plot. If all has gone to plan, it will look something like this:



In the case of a threshold filter report, the dashed line shows the threshold, and the legend identifies which data regions are selected by the different filters (in this case '0_below' or '1_above'). The reports for different types of filter are slightly different, and often include numerous groups of data. In this case, the 100 µmol/mol threshold seems to do a good job of excluding extraneously high Al/Ca values, so we'll use the '0_Al27_thresh_below' filter to select these data.

Applying a Filter

Once you've identified which filter you want to apply, you must turn that filter 'on' using:

```
eg.filter_on(filt='Albelow')
```

Where `filt` can either be the filter number (corresponding to the 'n' column in the output of `filter_status()`) or a partially matching string, as here. For example, `'Albelow'` is most similar to `'Al27_thresh_below'`, so this filter will be turned on. You could also specify `'below'`, which would turn on all filters with 'below' in the name. This is done using 'fuzzy string matching', provided by the `fuzzywuzzy` package. There is also a counterpart `eg.filter_off()` function, which works in the inverse. These functions will turn the threshold filter on for all analytes measured in all samples, and return a report of which filters are now on or off:

```
Subset All_Samples:
Samples: Sample-1, Sample-2, Sample-3

n  Filter Name          Mg24   Mg25   Al27   Ca43   Ca44   Mn55   Sr88   Ba137  Ba138
0  Al27_thresh_below    True   True   True   True   True   True   True   True   True
1  Al27_thresh_above    False  False  False  False  False  False  False  False  False
```

In some cases, you might have a sample where one analyte is effected by a contaminant that does not alter other analytes. If this is the case, you can switch a filter on or off for a specific analyte:

```
eg.filter_off(filt='Albelow', analyte='Mg25')
```

```
Subset All_Samples:
Samples: Sample-1, Sample-2, Sample-3

n  Filter Name          Mg24   Mg25   Al27   Ca43   Ca44   Mn55   Sr88   Ba137  Ba138
0  Al27_thresh_below    True   False  True   True   True   True   True   True   True
1  Al27_thresh_above    False  False  False  False  False  False  False  False  False
```

Notice how the 'Al27_thresh_below' filter is now deactivated for Mg25.

Deleting a Filter

When you create filters they are not automatically applied to the data (they are 'off' when they are created). This means that you can create as many filters as you want, without them interfering with each other, and then turn them on/off independently for different samples/analytes. There shouldn't be a reason that you'd need to delete a specific filter.

However, after playing around with filters for a while, filters can accumulate and get hard to keep track of. If this happens, you can use **:method:'latools.analyse.filter_clear'** to get rid of all of them, and then re-run the code for the filters that you like to re-create them.

Sample Subsets

Finally, let's return to the 'Subsets', which we skipped over earlier. It is quite common to analyse distinct sets of samples in the same analytical session. To accommodate this, you can create data 'subsets' during analysis, and treat them in different ways. For example, imagine that 'Sample-1' in our test dataset was a different type of sample, that needs to be filtered in a different way. We can identify this as a subset by:

```
eg.make_subset(samples='Sample-1', name='set1')
eg.make_subset(samples=['Sample-2', 'Sample-3'], name='set2')
```

And filters can be turned on and off independently for each subset:

```
eg.filter_on(filt=0, subset='set1')

Subset set1:
Samples: Sample-1

n  Filter Name          Mg24   Mg25   Al27   Ca43   Ca44   Mn55   Sr88   Ba137  Ba138
0  Al27_thresh_below    True   True   True   True   True   True   True   True   True
1  Al27_thresh_above    False  False  False  False  False  False  False  False  False


eg.filter_off(filt=0, subset='set2')

Subset set2:
Samples: Sample-2, Sample-3

n  Filter Name          Mg24   Mg25   Al27   Ca43   Ca44   Mn55   Sr88   Ba137  Ba138
0  Al27_thresh_below    False  False  False  False  False  False  False  False  False
1  Al27_thresh_above    False  False  False  False  False  False  False  False  False
```

To see which subsets have been defined:

```
eg.subsets

{'All_Analyses': ['Sample-1', 'Sample-2', 'Sample-3', 'STD-1', 'STD-2'],
 'All_Samples': ['Sample-1', 'Sample-2', 'Sample-3'],
 'STD': ['STD-1', 'STD-2'],
 'set1': ['Sample-1'],
 'set2': ['Sample-2', 'Sample-3']}
```

Note: The filtering above is relatively simplistic. More complex filters require quite a lot more thought and care in their application. For examples of how to use clustering, distribution and correlation filters, see the *Advanced Filtering*

section.

1.1.4.10 Sample Statistics

After filtering, you can calculated and export integrated compositional values for your analyses:

```
eg.sample_stats(stats=['mean', 'std'], filt=True)
```

Where `stats` specifies which functions you would like to use to calculate the statistics. Built in options are:

- `'mean'`: Arithmetic mean, calculated by `np.nanmean`.
- `'std'`: Arithmetic standard deviation, calculated by `np.nanstd`.
- `'se'`: Arithmetic standard error, calculated by `np.nanstd / n`.
- `'H15_mean'`: Huber (H15) robust mean.
- `'H15_std'`: Huber (H15) robust standard deviation.
- `'H15_se'`: Huber (H15) robust standard error.
- `custom_fn(a)`: A function you've written yourself, which takes an array (`a`) and returns a single value. This function must be able to cope with `NaN` values.

Where the Huber (H15) robust statistics remove outliers from the data, as described here.

You can specify any function that accepts an array and returns a single value here. `filt` can either be True (applies all active filters), or a specific filter number or partially matching name to apply a specific filter. In combination with data subsets, and the ability to specify different combinations of filters for different subsets, this provides a flexible way to explore the impact of different filters on your integrated values.

We've now calculated the statistics, but they are still trapped inside the 'analyse' data object (`eg`). To get them out into a more useful form:

```
stats = eg.getstats()
```

This returns a `pandas.DataFrame` containing all the statistics we just calculated. You can either keep this data in python and continue your analysis, or export the integrated values to an external file for analysis and plotting in *your_favourite_program*.

The calculated statistics are saved autoatmically to 'sample_stats.csv' in the 'data_export' directory. You can also specify the filename manually using the `filename` variable in `getstats()`, which will be saved in the 'data_export' directory, or you can use the pandas built in export methods like `to_csv()` or `to_excel()` to take your data straight to a variety of formats, for example:

```
stats.to_csv('stats.csv') # .csv format
```

1.1.4.11 Reproducibility

A key new feature of `latools` is making your analysis quantitatively reproducible. As you go through your analysis, `latools` keeps track of everything you're doing in a command log, which stores the sequence and parameters of every step in your data analysis. These can be exported, alongside an SRM table and your raw data, and be imported and reproduced by an independent user.

If you are unwilling to make your entire raw dataset available, it is also possible to export a 'minimal' dataset, which only includes the elements required for your analyses (i.e. any analyte used during filtering or processing, combined with the analytes of interest that are the focus of the reduction).

Minimal Export

The minimum parameters and data to reproduce you're analysis can be exported by:

```
eg.minimal_export()
```

This will create a new folder inside the `data_export` folder, called `minimal export`. This will contain your complete dataset, or a subset of your dataset containing only the analytes you specify, the SRM values used to calibrate your data, and a `.log` file that contains a record of everything you've done to your data.

This entire folder should be compressed (e.g. .zip), and included alongside your publication.

Tip: When someone else goes to reproduce your analysis, *everything* you've done to your data will be re-calculated. However, analysis is often an iterative process, and an external user does not need to experience *all* these iterations. We therefore recommend that after you've identified all the processing and filtering steps you want to apply to the data, you reprocess your entire dataset using *only* these steps, before performing a minimal export.

Import and Reproduction

To reproduce someone else's analysis, download a compressed minimal_export folder, and unzip it. Next, in a new python window, run:

```
import latools as la

rep = la.reproduce('path/to/analysis.log')
```

This will reproduce the entire analysis, and call it 'rep'. You can then experiment with different data filters and processing techniques to see how it modifies their results.

1.1.4.12 Summary

If we put all the preceding steps together:

```
eg = la.analyse(data_folder='./latools_demo_tmp',  # the location of your data
                config='UCD-AGILENT',  # the configuration to use
                internal_standard='Ca43',  # the internal standard in your analyses
                srm_identifier='STD')  # the text that identifies which files contain␣
↪standard reference materials
eg.trace_plots()

eg.despike(expdecay_despiker=True,
           noise_despiker=True)

eg.autorange(on_mult=[1.5, 0.8],
             off_mult=[0.8, 1.5])

eg.bkg_calc_weightedmean(weight_fwhm=300,
                         n_min=10)

eg.bkg_plot()

eg.bkg_subtract()
```

(continues on next page)

(continued from previous page)

```
eg.ratio()

eg.calibrate(drift_correct=False,
             srms_used=['NIST610', 'NIST612', 'NIST614'])

eg.calibration_plot()

eg.filter_threshold(analyte='Al27', threshold=100e-6)  # remember that all units are␣
↪in mol/mol!

eg.filter_reports(analytes='Al27', filt_str='thresh')

eg.filter_on(filt='Albelow')

eg.filter_off(filt='Albelow', analyte='Mg25')

eg.make_subset(samples='Sample-1', name='set1')
eg.make_subset(samples=['Sample-2', 'Sample-3'], name='set2')

eg.filter_on(filt=0, subset='set1')

eg.filter_off(filt=0, subset='set2')

eg.sample_stats(stats=['mean', 'std'], filt=True)

stats = eg.getstats()

eg.minimal_export()
```

Here we processed just 3 files, but the same procedure can be applied to an entire day of analyses, and takes just a little longer.

The processing stage most likely to modify your results is filtering. There are a number of filters available, ranging from simple concentration thresholds (`filter_threshold()`, as above) to advanced multi-dimensional clustering algorithms (`filter_clustering()`). We recommend you read and understand the section on advanced_filtering before applying filters to your data.

Before You Go

Before you try to analyse your own data, you must configure latools to work with your particular instrument/standards. To do this, follow the *Three Steps to Configuration* guide.

We also highly recommend that you read through the advanced_topics, so you understand how `latools` works before you start using it.

1.1.4.13 FAQs

I can't get my data to import...

Follow the instructions *here*. If you're really stuck,

Your software is broken. It doesn't work!

If you think you've found a bug in `latools` (i.e. not specific to your computer / Python installation), or that `latools` is doing something peculiar, we're keen to know about it. You can tell us about it by creating an issue on the project GitHub page. Describe the problem as best you can, preferably with some examples, and we'll get to it as soon as we can.

I want to do X, can you add this feature?

Probably! Head on over to the GitHub project page, and create an issue. Write us a detailed description of what you're trying to do, and label the issue as an 'Enhancement' (on the right hand side), and we'll get to it as soon as we can.

1.1.5 Example Analyses

1. Cultured foraminifera data, and comparison to manually reduced data.
2. Downcore (fossil) foraminifera data, and comparison to manually reduced data.
3. Downcore (fossil) foraminifera data, and comparison to data reduced with Iolite.
4. Zircon data, and comparison to values reported in Burnham and Berry, 2017.

All these notebooks and associated data are available for download here.

1.1.6 Filters

These pages contain specific information about the types of filters available in `latools`, and how to use them.

For a general introduction to filtering, head over to the *Data Selection and Filtering* section of the Begginer's Guide.

1.1.6.1 Thresholds

Thresholds are the simplest type of filter in `latools`. They identify regions where the concentration or local gradient of an analyte is above or below a threshold value.

Appropriate thresholds may be determined from prior knowledge of the samples, or by examining whole-analysis level cross-plots of the concentration or local gradients of all pairs of analytes, which reveal relationships within all the ablations, allowing distinct contaminant compositions to be identified and removed.

Tip: All the following examples will work on the example dataset worked through in the *Beginner's Guide*. If you try multiple examples, be sure to run `eg.filter_clear()` in between each example, or the filters might not behave as expected.

Concentration Filter

Selects data where a target analyte is above or below a specified threshold.

For example, applying an threshold Al/Ca value of 100 μmol/mol to `Sample-1` of the example data:

```
# Create the filter.
eg.filter_threshold(analyte='Al27', threshold=100e-6)
```

This creates two filters - one that selects data above the threshold, and one that selects data below the threshold. To see what filters you've created, and whether they're 'on' or 'off', use *filter_status()*, which will print:

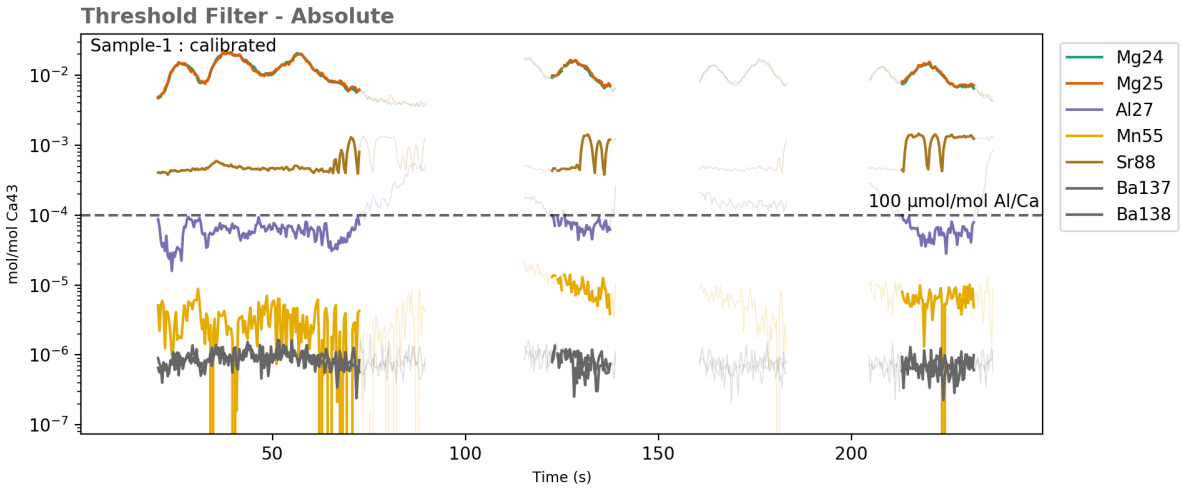
```
Subset All_Samples:

n  Filter Name          Mg24   Mg25   Al27   Ca43   Ca44   Mn55   Sr88   Ba137  Ba138
0  Al27_thresh_below    False  False  False  False  False  False  False  False  False
1  Al27_thresh_above    False  False  False  False  False  False  False  False  False
```

To effect the data, a filter must be activated:

```
# Select data below the threshold
eg.filter_on('Al27_below')

# Plot the data for Sample-1 only
eg.data['Sample-1'].tplot(filt=True)
```



Data above the threshold values (dashed line) are excluded by this filter (greyed out).

Tip: When using *filter_on()* or *filter_off()*, you don't need to specify the *entire* filter name displayed by *filter_status()*. These functions identify the filter with the name most similar to the text you entered, and activate/deactivate it.

Related Functions

- *filter_threshold()* creates a threshold filter.
- *filter_on()* and *filter_off()* turn filters on or off.
- *crossplot()* creates a cross-plot of all analytes, showing relationships within the data at the population-level (all samples). This can be useful when choosing a threshold value.
- *filter_reports()* creates plots of a particular filter, showing which sections of the ablation are selected.
- *histograms()* creates histograms of the concentrations of all analytes. Useful for identifying threshold values for specific analytes.
- *trace_plots()* with option `filt=True` creates plots of all data, showing which regions are selected/rejected by the active filters.

- *filter_clear()* deletes all filters.

Gradient Filter

Selects data where a target analyte is not changing - i.e. its gradient is constant. This filter starts by calculating the local gradient of the target analyte:

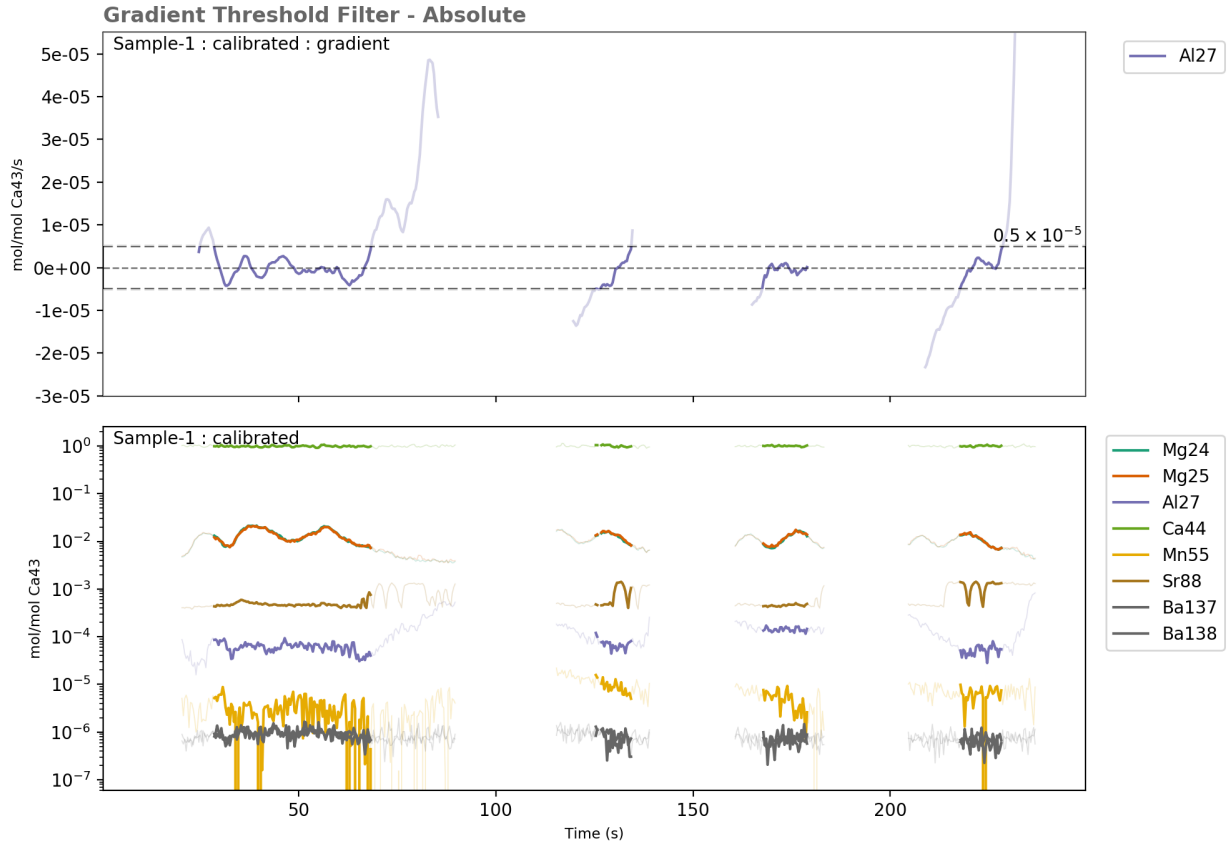
Fig. 1: Calculating a moving gradient for the Al27 analyte. When calculating the gradient the `win` parameter specifies how many points are used when calculating the local gradient.

For example, imagine a calcium carbonate sample which we know should have constant Al concentration. In this sample, variable Al is indicative of a contaminant phase. A gradient threshold filter can be used to isolate regions where Al is constant, and more likeley to be contaminant-free. To create an apply this filter:

```
eg.filter_gradient_threshold(analyte='Al27', threshold=0.5e-5, win=25)

eg.filter_on('Al27_g_below')

# plot the gradient for Sample-1
eg.data['Sample-1'].gplot('Al27', win=25)
# plot the effect of the filter for Sample-1
eg.data['Sample-1'].tplot(filt=True)
```



The top panel shows the calculated gradient, with the regions above and below the threshold value greyed out. the bottom panel shows the data regions selected by the filter for all elements.

Choosing a gradient `threshold` value

Gradients are in units of **mol[X] / mol[internal standard] / s**. The absolute value of the gradient will change depending on the value of `win` used.

Working out what a gradient `threshold` value should be from first principles can be a little complex. The best way to choose a threshold value is by looking at the data. There are three functions to help you do this:

- *`gradient_plots()`* Calculates the local gradient of all samples, plots the gradients, and saves them as a pdf. The gradient equivalent of *`trace_plots()`*.
- *`gradient_histogram()`* Plot histograms of the local gradients in the entire dataset.
- *`gradient_crossplot()`* Create crossplots of the local gradients for all analyes.

Tip: The value of `win` used when calculating the gradient will effect the absolute value of the calculated gradient. Make sure you use the same `win` value creating filters and viewing gradients.

Related Functions

- *`filter_threshold()`* creates a threshold filter.
- *`filter_on()`* and *`filter_off()`* turn filters on or off.
- *`gradient_plots()`* Calculates the local gradient of all samples, plots the gradients, and saves them as a pdf. The gradient equivalent of *`trace_plots()`*.
- *`gradient_crossplot()`* Create crossplots of the local gradients for all analyes.
- *`gradient_histogram()`* Plot histograms of the local gradients in the entire dataset.
- *`trace_plots()`* with option `filt=True` creates plots of all data, showing which regions are selected/rejected by the active filters.
- *`filter_reports()`* creates plots of a particular filter, showing which sections of the ablation are selected.
- *`filter_clear()`* deletes all filters.

1.1.6.2 Percentile Thresholds

In cases where the absolute threshold value is not known, a percentile may be used. An absolute threshold value is then calculated from the raw data at either the individual-ablation or population level, and used to create a threshold filter.

Warning: In general, we discourage the use of percentile filters. It is always better to examine and understand the patterns in your data, and choose absolute thresholds. However, we have come across cases where they have proved useful, so they remain an available option.

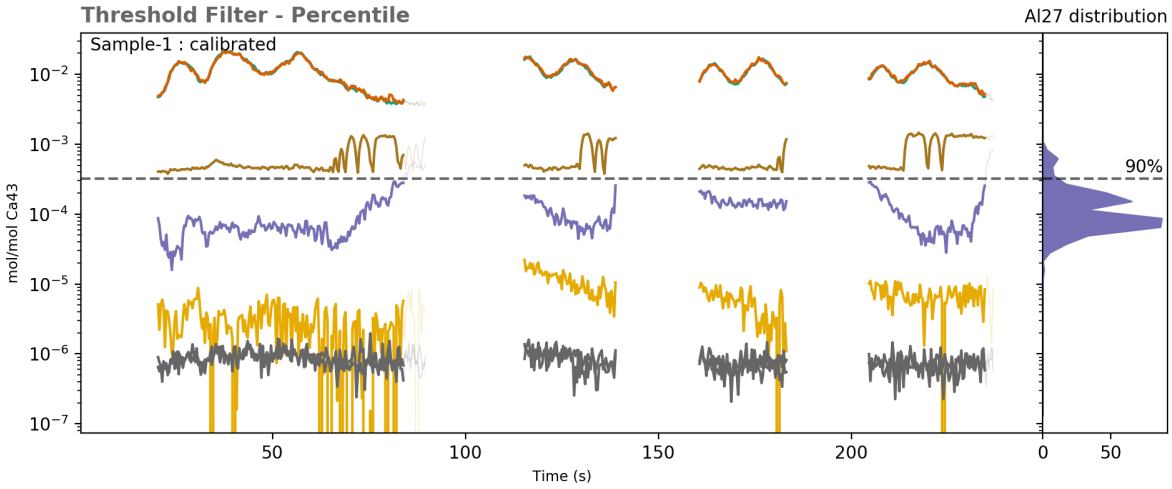
Concentration Filter: Percentile

For example, to remove regions containing the top 10% of Al concentrations:

```
eg.filter_threshold_percentile(analyte='Al27', percentiles=90)

eg.filter_on('Al_below')

eg.data['Sample-1'].tplot(filt=True)
```



The histogram on the right shows the distribution of Al data in the sample, with a line showing the 90th percentile of the data, corresponding to the threshold value used.

Gradient Filter: Percentile

The principle of this filter is the same, but it operatures on the local gradient of the data, instead of the absolute concentrations.

1.1.6.3 Correlation

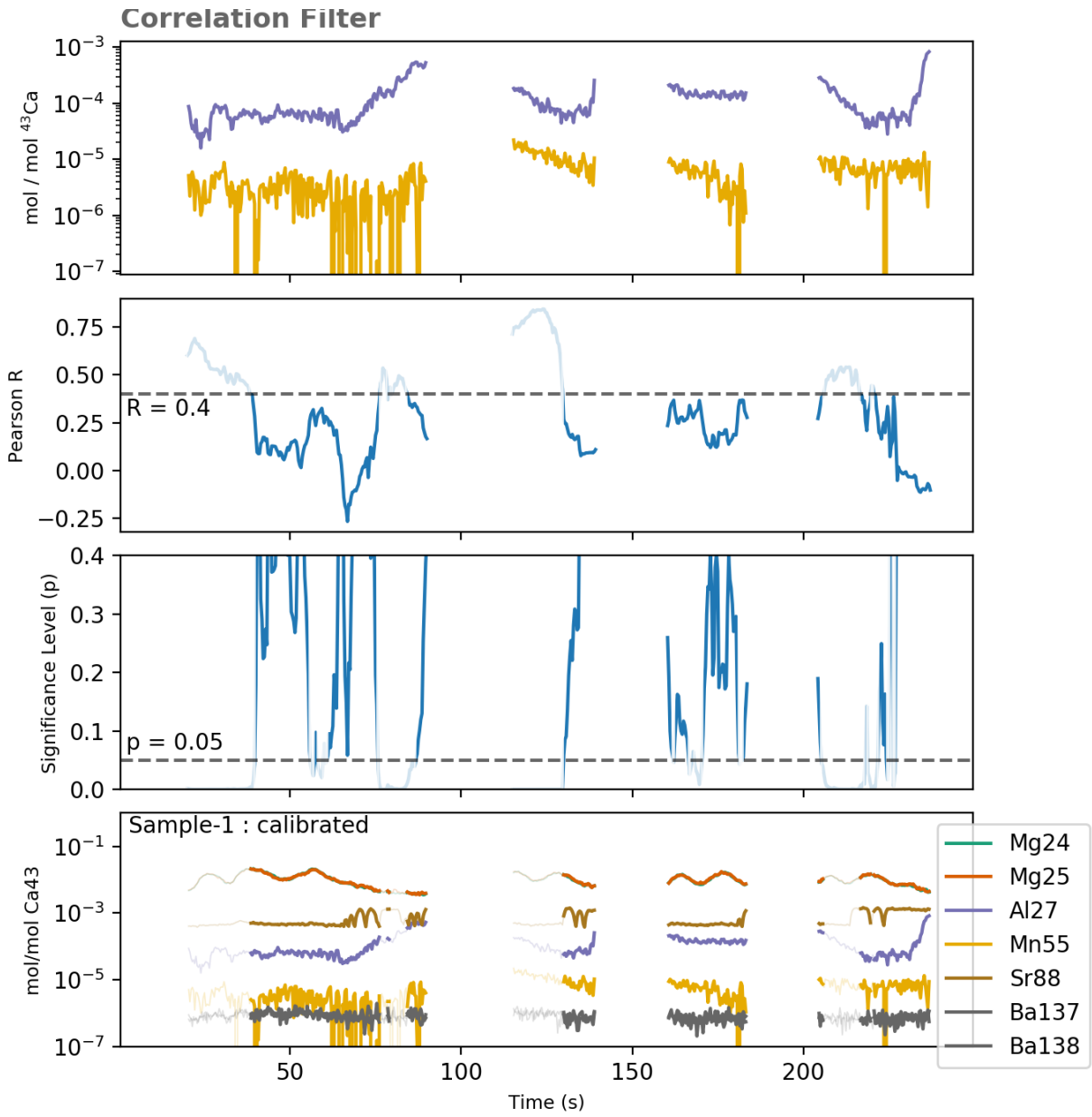
Correlation filters identify regions in the signal where two analytes increase or decrease in tandem. This can be useful for removing ablation regions contaminated by a phase with similar composition to the host material, which influences more than one element.

For example, the tests of foraminifera (biomineral calcium carbonate) are known to be relatively homogeneous in Mn/Ca and Al/Ca. When preserved in marine sediments, the tests can become contaminated with clay minerals that are enriched in Mn and Al, and unknown concentrations of other elements. Thus, regions where Al/Ca and Mn/Ca co-vary are likely contaminated by clay materials. A Al vs. Mn correlation filter can be used to exclude these regions.

For example:

```
eg.filter_correlation(x_analyte='Al27', y_analyte='Mn55', window=51, r_threshold=0.5,␣
↪p_threshold=0.05)
eg.filter_on('AlMn')

eg.data['Sample-1'].tplot(filt=True)
```



The top panel shows the two target analytes. The Pearson correlation coefficient (R) between these elements, along with the significance level of the correlation (p) is calculated for 51-point rolling window across the data. Data are excluded in regions R is greater than `r_threshold` and p is less than `p_threshold`.

The second panel shows the Pearson R value for the correlation between these elements. Regions where R is above the `r_threshold` value are excluded.

The third panel shows the significance level of the correlation (p). Regions where p is less than `p_threshold` are excluded.

The bottom panel shows data regions excluded by the combined R and p filters.

Choosing R and p thresholds

The pearson R value ranges between -1 and 1, where 0 is no correlation, -1 is a perfect negative, and 1 is a perfect positive correlation. The R values of the data will be effected by both the degree of correlation between the analytes, and the noise in the data. Choosing an absolute R threshold is therefore not straightforward.

Tip: The filter does not discriminate between positive and negative correlations, but considers the absolute R value - i.e. an `r_threshold` of 0.9 will remove regions where R is greater than 0.9, and less than -0.9.

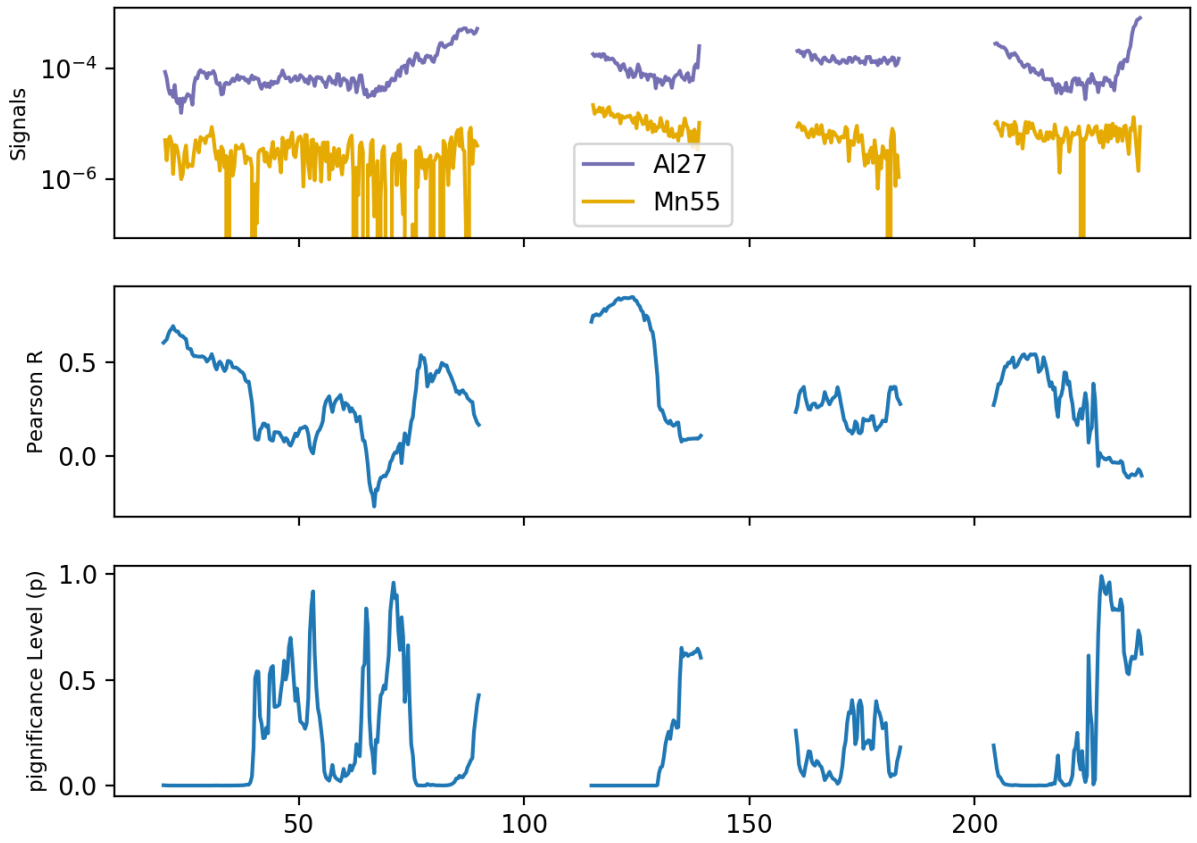
Similarly, the p value of the correlation will depend on the strength of the correlation, the window size used, and the noise in the data.

The best way to choose thresholds is by looking at the correlation values, using *`correlation_plots()`* to inspect inter-analyte correlations before creating the filter.

For example:

```
eg.correlation_plots(x_analyte='Al27', y_analyte='Mn55', window=51)
```

Will produce pdf plots like the following for all samples.



Related Functions

- *`correlation_plots()`* creates plots of the local correlation between two analytes.

- *crossplot()* creates a cross-plot of all analytes, showing relationships within the data at the population-level (all samples). This can be useful when choosing a threshold value.
- *trace_plots()* with option filt=True creates plots of all data, showing which regions are selected/rejected by the active filters.
- *filter_on()* and *filter_off()* turn filters on or off.
- *filter_clear()* deletes all filters.

1.1.6.4 Clustering

The clustering filter provides a convenient way to separate compositionally distinct materials within your ablations, using multi-dimensional clustering algorithms.

Two algorithms are currently available in latools: * K-Means will divide the data up into N groups of equal variance, where N is a known number of groups. * Mean Shift will divide the data up into an arbitrary number of clusters, based on the characteristics of the data.

For an in-depth explanation of these algorithms and how they work, take a look at the Scikit-Learn clustering pages.

For most cases, we recommend the K-Means algorithm, as it is relatively intuitive and produces more predictable results.

2D Clustering Example

For illustrative purposes, consider some 2D synthetic data:

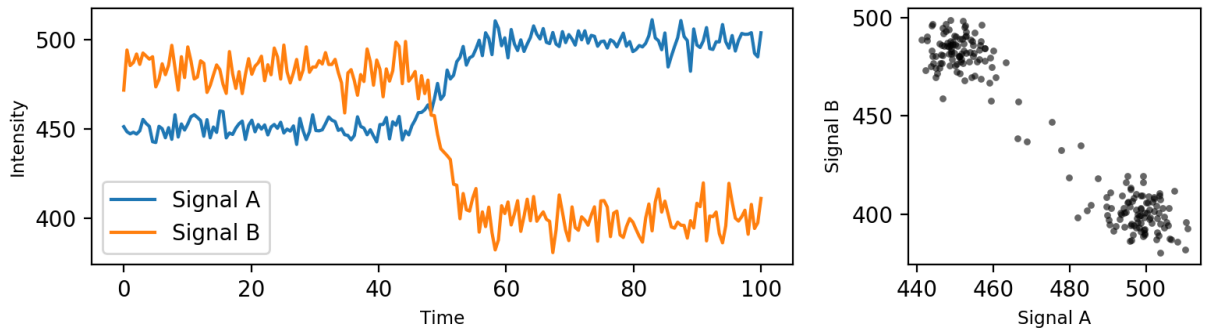


Fig. 2: The left panel shows two signals (A and B) which transition from an initial state where B > A (<40 s) to a second state where A > B (>60 s). In laser ablation terms, this might represent a change in concentration of two analytes at a material boundary. The right panel shows the relationship between the A and B signals, ignoring the time axis.

Two 'clusters' in composition are evident in the data, which can be separated by clustering algorithms.

The main difference here is that the MeanShift algorithm has identified the transition points (orange) as a separate cluster.

Once the clusters are identified, they can be translated back into the time-domain to separate the signals in the original data:

For simplicity, the example above considers the relationship between two signals (i.e. 2-D). When creating a clustering filter on real data, multiple analytes may be included (i.e. N-D). The only limits on the number of analytes you can include is the number of analytes you've measured, and how much RAM your computer has.

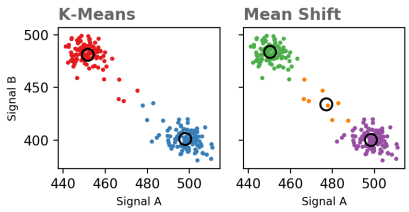


Fig. 3: In the left panel, the K-Means algorithm has been used to find the boundary between two distinct materials. In the right panel, the Mean Shift algorithm has automatically detected three materials.

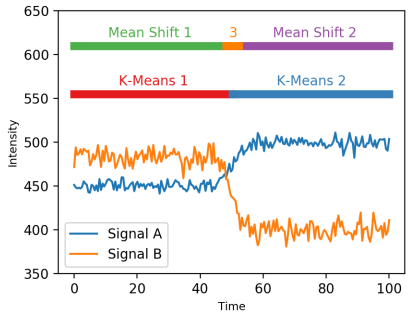


Fig. 4: Horizontal bars denote the regions identified by the K-Means and MeanShift clustering algorithms.

If, for example, your ablation contains three distinct materials with variations in five analytes, you might create a K-Means clustering filter that takes all five analytes, and separates them into three clusters.

When to use a Clustering Filter

Clustering filters should be used to discriminate between clearly different materials in an analysis. Results will be best when they are based on signals with clear sharp changes, and high signal/noise (as in the above example).

Results will be poor when data are noisy, or when the transition between materials is very gradual. In these cases, clustering filters may still be useful after you have used other filters to remove the transition regions - for example gradient-threshold or correlation filters.

Clustering Filter Design

A good place to start when creating a clustering filter is by looking at a cross-plot of your analytes:

```
eg.crossplot()
```

A crossplot provides an overview of your data, and allows you to easily identify relationships between analytes. In this example, multiple levels of Sr88 concentration are evident, which we might want to separate. Three Sr88 groups are evident, so we will create a K-Means filter with three clusters:

```
eg.filter_clustering(analyte='Sr88', level='population', method='kmeans', n_
↪clusters=3)

eg.filter_status()
```

(continues on next page)

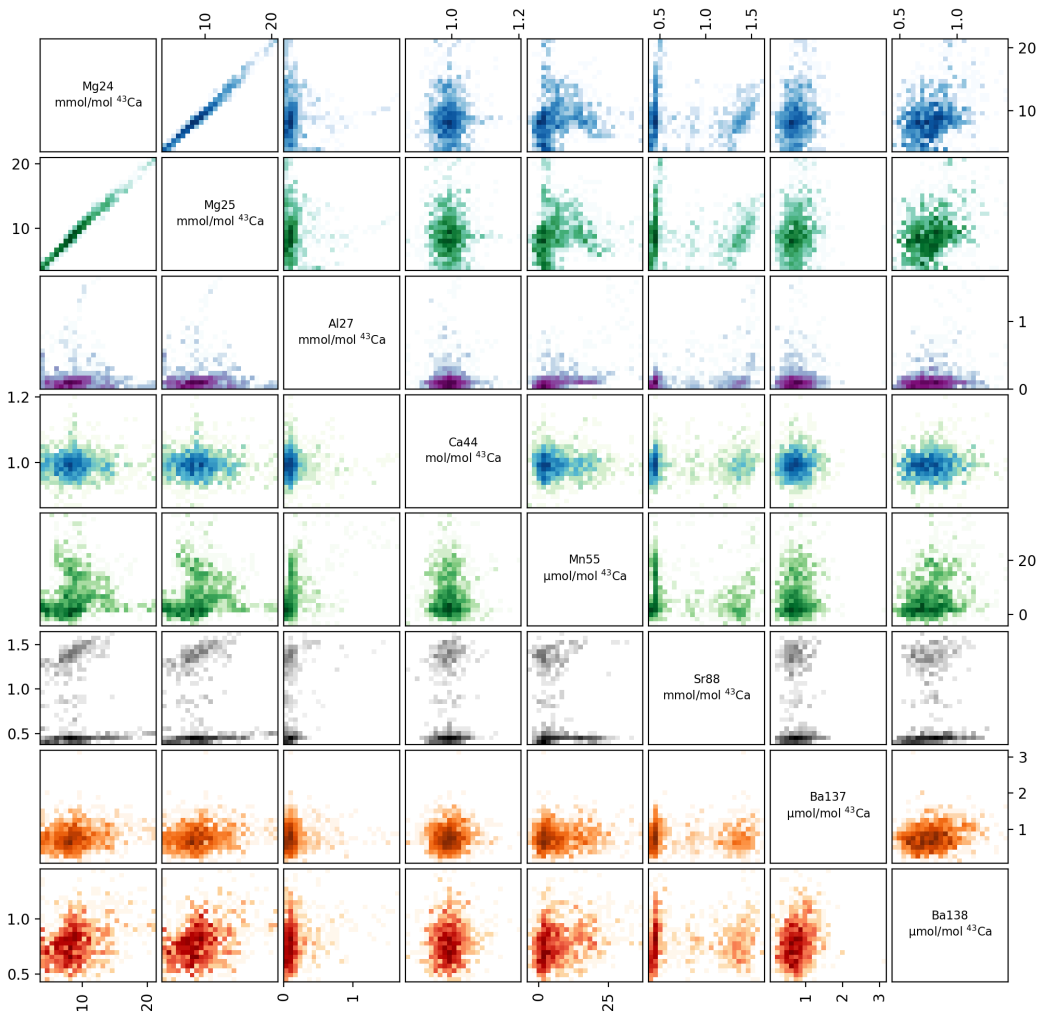


Fig. 5: A crossplot showing relationships between all measured analytes in all samples. Data are presented as 2D histograms, where the intensity of colour relates to the number of data points in that pixel. In this example, a number of clusters are evident in both Sr88 and Mn55, which are candidates for clustering filters.

(continued from previous page)

```
> Subset: 0
> Samples: Sample-1, Sample-2, Sample-3
>
> n  Filter Name      Mg24   Mg25   Al27   Ca43   Ca44   Mn55   Sr88   Ba137  Ba138
> 0  Sr88_kmeans_0    False  False  False  False  False  False  False  False  False
> 1  Sr88_kmeans_1    False  False  False  False  False  False  False  False  False
> 2  Sr88_kmeans_2    False  False  False  False  False  False  False  False  False
```

The clustering filter has used the population-level data to identify three clusters in Sr88 concentration, and created a filter based on these concentration levels.

We can directly see the influence of this filter:

```
eg.crossplot_filters('Sr88_kmeans')
```

Tip: You can use `crossplot_filter` to see the effect of any created filters - not just clustering filters!

Here, we can see that the filter has picked out three Sr concentrations well, but that these clusters don't seem to have any systematic relationship with other analytes. This suggests that Sr might not be that useful in separating different materials in these data. (In reality, the Sr variance in these data comes from an incorrectly-tuned mass spec, and tells us nothing about the sample!)

Related Functions

- *`crossplot()`* creates a cross-plot of specified analytes, showing relationships within the data at the population-level (all samples). This can be useful when choosing a threshold value.
- *`crossplot_filters()`* creates a cross-plot of specified analytes with the effect of a particular filter highlighted (see above).
- *`trace_plots()`* with option `filt=True` creates plots of all data, showing which regions are selected/rejected by the active filters.
- *`filter_on()`* and *`filter_off()`* turn filters on or off.
- *`filter_clear()`* deletes all filters.

1.1.6.5 Signal Optimisation

This is the most complex filter available within `latools`, but can produce some of the best results.

The filter aims to identify the longest contiguous region within each ablation where the concentration of target analyte(s) is either maximised or minimised, and standard deviation is minimised.

First, we calculate the mean and standard deviation for the target analyte over all sub-regions of the data.

Next, we use the distributions of the calculated means and standard deviations to define some threshold values to identify the optimal region to select. For example, if the goal is to minimise the concentration of an analyte, the threshold concentration value will be the lowest disinct peak in the histogram of region means. The location of this peak defines the 'mean' threshold. Similarly, as the target is always to minimise the standard deviation, the standard deviation threshold will also be the lowest distinct peak in the histogram of standard devaiations of all calculated regions. Once identified, these thresholds are used to 'filter' the calculated means and standard deviations. The 'optimal' selection has a mean and standard deviation below the calculated threshold values, and contains the maximum possible number of data points.

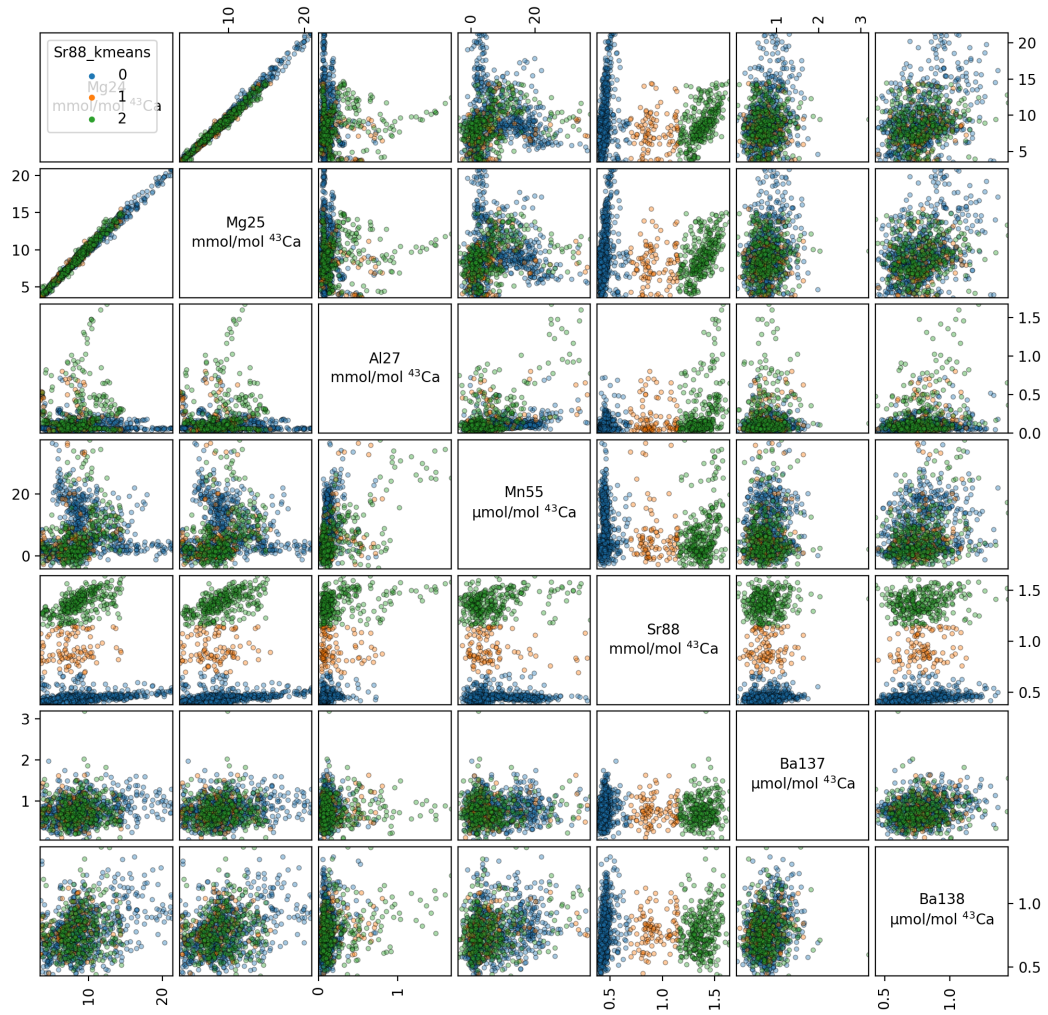


Fig. 6: A crossplot of all the data, highlighting the clusters identified by the filter.

Fig. 7: The mean and standard devation of Al27 is calculated using an N-point rolling window, then N+1, N+2 and etc., until N equals the number of data points. In the 'Mean' plot, darker regions contain the lowest values, and in the 'Standard Deviation' plot red regions contain a higher standard deviation.

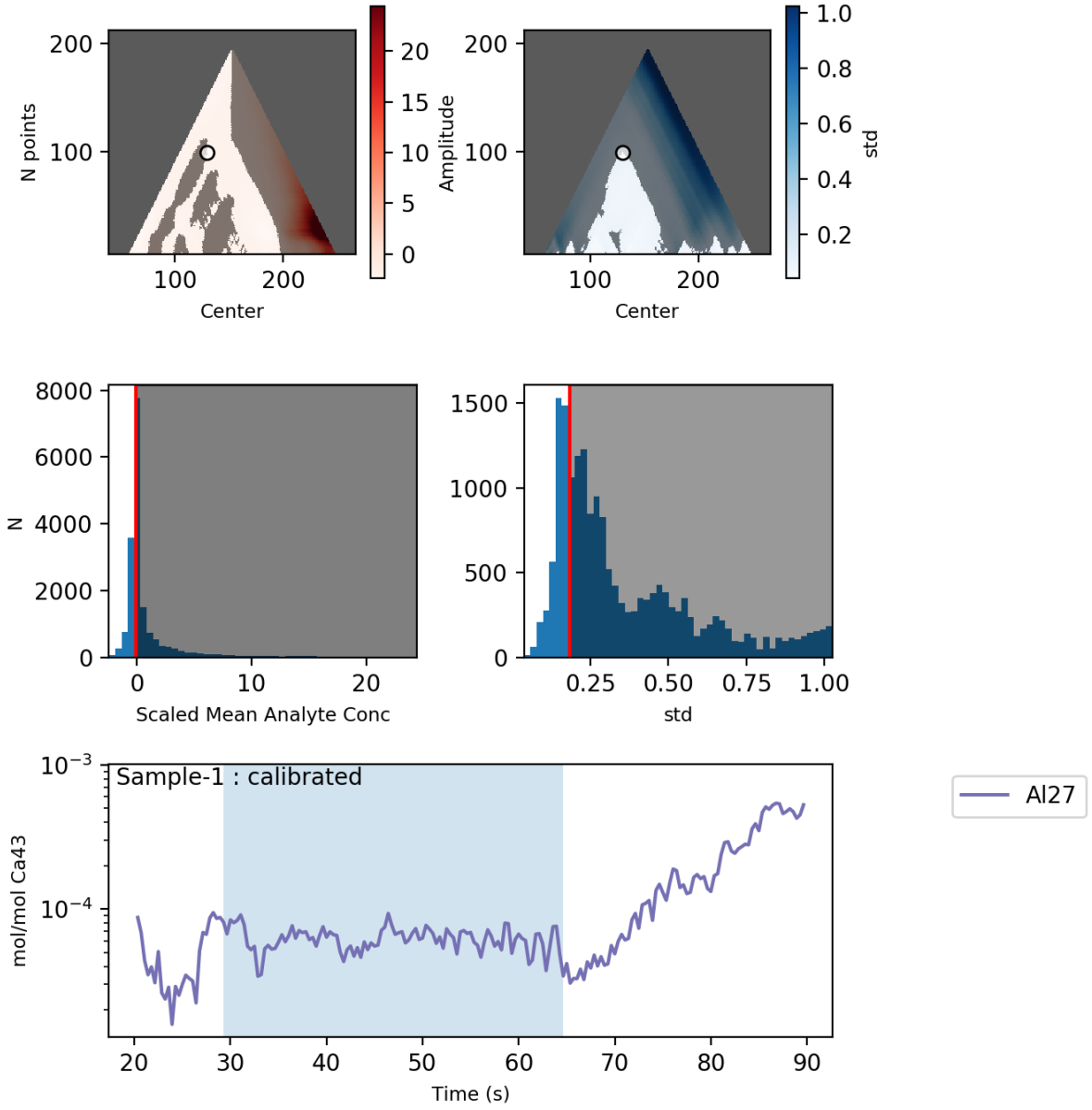


Fig. 8: After calculating the mean and standard deviation for all regions, the optimal region is identified using threshold values derived from the distributions of sub-region means and standard deviations. These thresholds are used to 'filter' the calculated means and standard deviations - regions where they are above the threshold values are greyed out in the top row of plots. The optimal selection is the largest region where both the standard deviation and mean are below the threshold values.

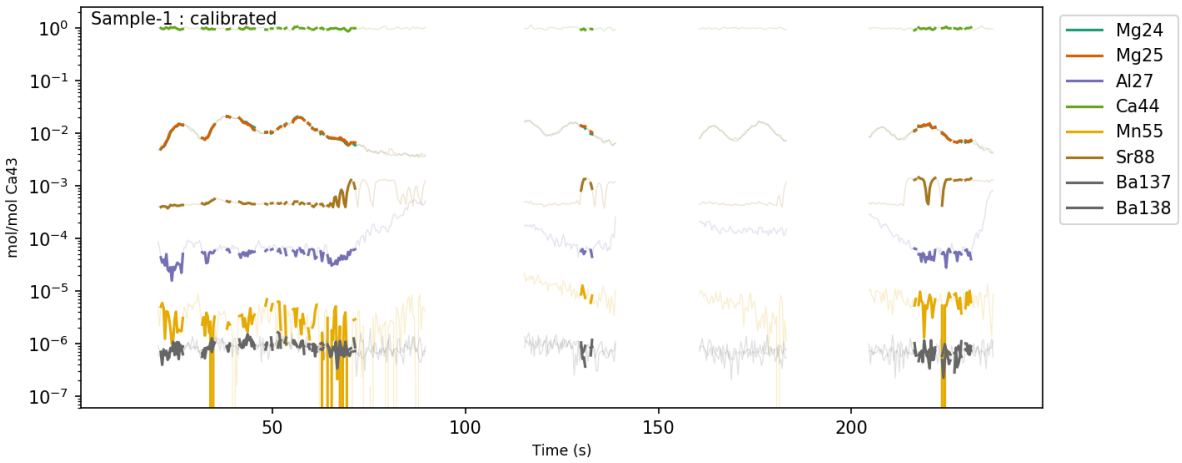
Related Functions

- *optimisation_plots()* creates plots similar to the one above, showing the action of the optimisation algorithm.
- *trace_plots()* with option `filt=True` creates plots of all data, showing which regions are selected/rejected by the active filters.
- *filter_on()* and *filter_off()* turn filters on or off.
- *filter_clear()* deletes all filters.

1.1.6.6 Defragmentation

Occasionally, filters can become 'fragmented' and erroneously omit or include lots of small data fragments. For example, if a signal oscillates either side of a threshold value. The defragmentation filter provides a way to either include incorrectly removed missing data regions, or exclude data in fragmented regions.

```
eg.filter_threshold('Al27', 0.65e-4)
eg.filter_on('Al27_below')
```

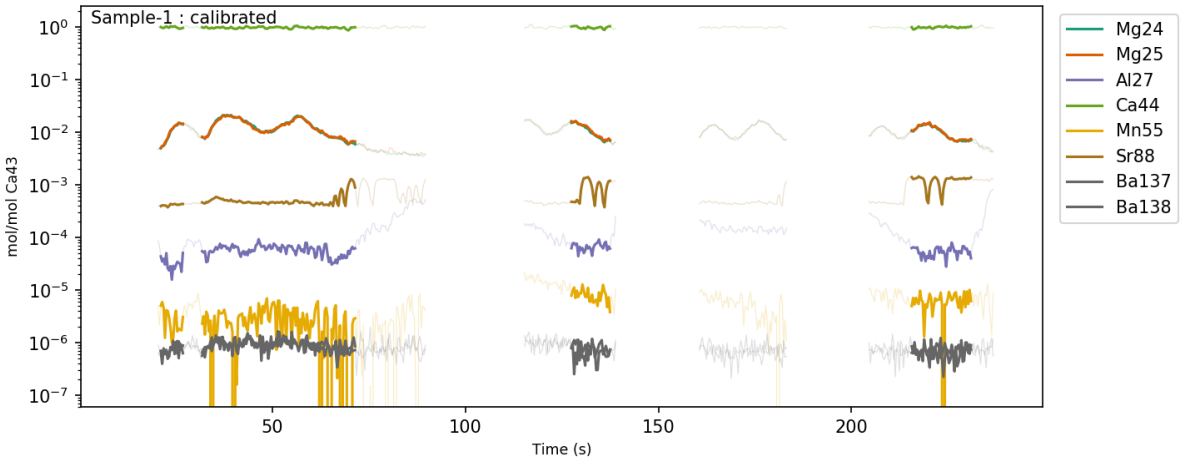


Notice how this filter has removed lots of small data regions, where Al27 oscillates around the threshold value.

If you think these regions should be included in the selection, the defragmentation filter can be used in 'include' mode to create a contiguous data selection:

```
eg.filter_defragment(10, mode='include')

eg.filter_off('Al27')  # deactivate the original Al filter
eg.filter_on('defrag')  # activate the new defragmented filter
```



This identifies all regions removed by the currently active filters that are 10 points or less in length, and includes them in the data selection.

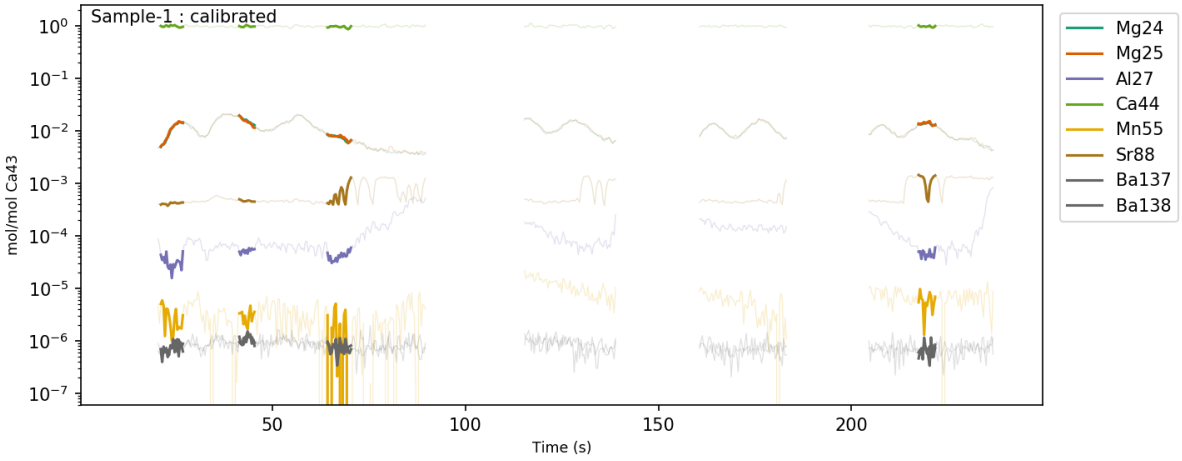
Tip: The defragmentation filter acts on all currently active filters, so pay attention to which filters are turned 'on' or 'off' when you use it. You'll also need to de-activate the filters used to create the defragmentation filter to see its effects.

If, on the other hand, the proximity of Al27 in this sample to the threshold value might suggest contamination, you can use 'exclude' mode to remove small regions of selected data.

```
eg.filter_threshold('Al27', 0.65e-4)
eg.filter_on('Al27_below')


eg.filter_defragment(10, mode='exclude')

eg.filter_off()  # deactivate the original Al filter
eg.filter_on('defrag')  # activate the new defragmented filter
```



This removes all fragments fragments of selected data that are 10-points or less, and removes them.

Related Functions

- *trace_plots()* with option `filt=True` creates plots of all data, showing which regions are selected/rejected by the active filters.
- *filter_on()* and *filter_off()* turn filters on or off.
- *filter_clear()* deletes all filters.

1.1.6.7 Down-Hole Exclusion

This filter is specifically designed for spot analyses were, because of side-wall ablation effects, data collected towards the end of an ablation will be influenced by data collected at the start of an ablation.

This filter provides a means to exclude all material 'down-hole' of the first excluded contaminant.

For example, to continue the example from the *Defragmentation* Filter, you may end up with a selection that looks like this:

In the first ablation of this example, the defragmentation filter has left four data regions selected. Because of down-hole effects, data in the second, third and fourth regions will be influenced by the material ablated at the start of the sample. If there is a contaminant at the start of the sample, this contaminant will also have a minor influence on these regions, and they should be excluded. This can be done using the Down-Hole Exclusion filter:

```
eg.filter_exclude_downhole(threshold=5)
# threshold sets the number of consecutive excluded points after which
# all data should be excluded.

eg.filter_off()
eg.filter_on('downhole')
```

This filter is particularly useful if, for example, there is a significant contaminated region in the middle of an ablation, but threshold filters do not effectively exclude the post-contaminant region.

Related Functions

- *trace_plots()* with option `filt=True` creates plots of all data, showing which regions are selected/rejected by the active filters.
- *filter_on()* and *filter_off()* turn filters on or off.
- *filter_clear()* deletes all filters.

1.1.6.8 Trimming/Expansion

This either expands or contracts the currently active filters by a specified number of points.

Trimming a filter can be a useful tool to make selections more conservative, and more effectively remove contaminants.

Related Functions

- *trace_plots()* with option `filt=True` creates plots of all data, showing which regions are selected/rejected by the active filters.
- *filter_on()* and *filter_off()* turn filters on or off.
- *filter_clear()* deletes all filters.

1.1.7 Preprocessing

`latoools` expects data to be organised in a *particular way*. If your data do not meet these expectations, you'll have to do some pre-processing to get your data into a format that latools can deal with. These pages contain information on the 'preprocessing' tools you can use to prepare your data for `latools`.

If the methods described here don't work for you, or your data is in a format that can't be handled by them, please let us know and we'll work out how to accommodate your data.

1.1.7.1 Long File Splitting

If you've collected data from ablations of multiple samples and standards in a single, long data file, read on.

To work with this data, you have to split it up into numerous shorter files, each containing ablations of a single sample. This can be done using *latools.preprocessing.split.long_file()*.

Ingredients

- A single data file containing multiple analyses
- A *Data Format description* for that file (you can also use pre-configured formats).
- A list of names for each ablation in the file.

To keep things organise, we suggest creating a file structure like this:

```
my_analysis/
    my_long_data_file.csv
    sample_list.txt
```

Tip: In this example we've shown the sample list as a text file. It can be in any format you want, as long as you can import it into python and turn it into a list or array to give it to the splitter function.

Method

1. Import your data, and provide a list of sample names.
2. Apply *autorange()* to identify ablations.
3. Match the sample names up to the ablations.
4. Save a single file for each sample in an output folder, which can be imported by *analyse()*
5. Plot a graph showing how the file has been split, so you can make sure everything has worked as expected.

Output

After you've applied *long_file()*, a few more files will have been created, and your directory structure will look like this:

```
my_analysis/
    my_long_data_file.csv
    sample_list.txt
    my_long_data_file_split/
        STD_1.csv
        STD_2.csv
        Sample_1.csv
        Sample_2.csv
        Sample_3.csv
        ... etc.
```

If you have multiple consecutive ablations with the same name (i.e. repeat ablations of the same sample) these will be saved to a single file that contains all the ablations of the same file.

Example

To try this example at home `this zip file` contains all the files you'll need.

Unzip this file, and you should see the following files:

```
long_example/
    long_data_file.csv  # the data file
    long_data_file_format.json  # the format of that file
    long_example.ipynb  # a Jupyter notebook containing this example
    sample_list.txt  # a list of samples in plain text format
    sample_list.xslx  # a list of samples in an Excel file.
```

1. Load Sample List

First, read in the list of samples in the file. We have examples in two formats here - both plain text and in an Excel file. We don't care what format the sample list is in, as long as you can read it in to Python as an array or a list. In the case of these examples:

Text File

```
import numpy as np
sample_list = np.genfromtxt('long_example/sample_list.txt',  # read this file
                            dtype=str,  # the data are in text ('string') format
                            delimiter='\n',  # separated by new-line characters
                            comments='#'  # and lines starting with # should be␣
↪ignored.
                            )
```

This loads the sample list into a numpy array, which looks like this:

```
array(['NIST 612', 'NIST 612', 'NIST 610', 'jcp', 'jct', 'jct',
       'Sample_1', 'Sample_1', 'Sample_1', 'Sample_1', 'Sample_1',
       'Sample_2', 'Sample_2', 'Sample_2', 'Sample_3', 'Sample_3',
       'Sample_3', 'Sample_4', 'Sample_4', 'Sample_4', 'Sample_5',
       'Sample_5', 'Sample_5', 'Sample_5', 'Sample_5', 'Sample_5',
       'NIST 612', 'NIST 612', 'NIST 610', 'jcp', 'jct', 'jct'],
      dtype='<U8')
```

Excel File

```
import pandas as pd
sample_list = pd.read_excel('long_example/sample_list.xlsx')
```

This will load the data into a DataFrame, which looks like this:

The sample names can be accessed using:

```
sample_list.loc[:, 'Samples']
```

2. Split the Long File

```
import latools as la

fig, ax = la.preprocessing.long_file('long_example/long_data_file.csv',
                                     dataformat='long_example/long_data_file_format.
↪json',
                                     sample_list=sample_list.loc[:, 'Samples'])  #␣
↪note we're using the excel file here.
```

This will produce some output telling you what it's done:

The single long file has been split into 13 component files in the format that `latools` expects - each file contains ablations of a single sample. Note that consecutive ablations with the same sample are combined into single files, and if a sample name is repeated `_N` is appended to the sample name, to make the file name unique.

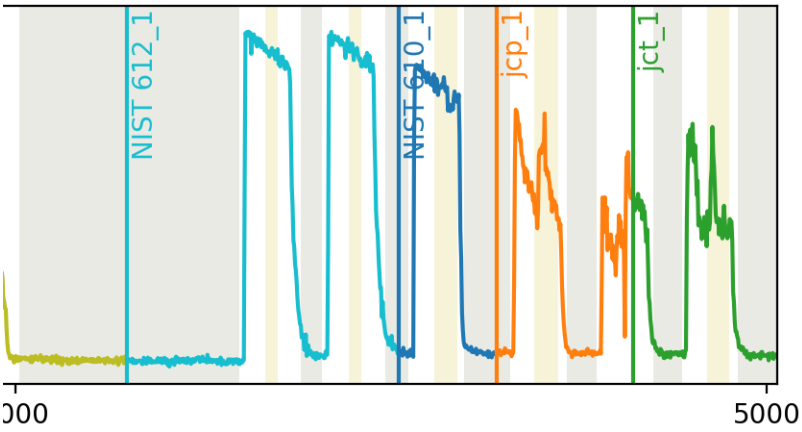
The function also produces a plot showing how it has split the files:

3. Check Output

So far so good, right? **NO!** This split has not worked properly.

Take a look at the printed output. On the second line, it says that the number of samples in the list and the number of ablations don't match. This is a red flag - either your sample list is wrong, or latools is not correctly identifying the number of ablations.

The key to diagnosing these problems lies in the plot showing how the file has split the data. Take a look at the right hand side of this plot:



Something has gone wrong with the separation of the `jcp` and `jct` ablations. This is most likely related to the signal decreasing to close to zero mid-way through the the second-to-last ablation, causing it to be itendified as two separate ablations.

4. Troubleshooting

In this case, a simple solution could be to smooth the data before splitting.

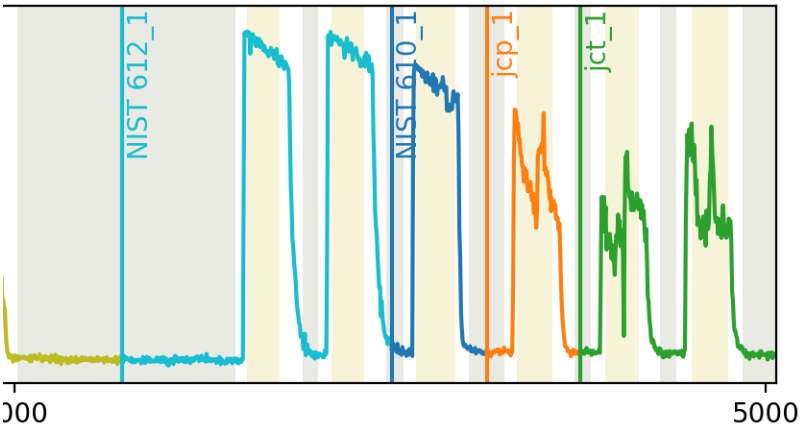
The *`long_file()`* function uses *`autorange()`* to identify ablations in a file, and you can modify any of the autorange parameters by passing giving them directly to *`long_file()`*.

Take a look at the *`autorange()`* documentation. Notice how the input parameter `swin` applies a smoothing window to the data before the signal is processed. So, to smooth the data before splitting it, we can simply add an `swin` argument to *`long_file()`*:

```
fig, ax = la.preprocessing.long_file('long_example/long_data_file.csv',
                                     dataformat='long_example/long_data_file_format.
↪json',
                                     sample_list=sample_list.loc[:, 'Samples'],
                                     swin=10)  # I'm using 10 here because it seems␣
↪to work well... Pick whatever value works for you.
```

This produces the output:

You can see in the image that this has fixed the issue:



5. Analyse

You can now continue with you `latools` analysis, as normal.

```
dat = la.analyse('long_atom/10454_TRA_Data_split', config='REPRODUCE', srm_identifier=
↪'NIST')
dat.despike()
dat.autorange(off_mult=[1, 4.5])
dat.bkg_calc_weightedmean(weight_fwhm=1200)
dat.bkg_plot()
dat.bkg_subtract()
dat.ratio()
dat.calibrate(srms_used=['NIST610', 'NIST612'])
_ = dat.calibration_plot()

# and etc...
```

1.1.8 Configuration Guide

1.1.8.1 Three Steps to Configuration

Warning: `latools` **will not work** if incorrectly configured. Follow the instructions below carefully.

Like all software, `latools` is stupid. It won't be able to read your data or know the composition of your reference materials, unless you tell it how.

1. Data Format Description

Mass specs produce a baffling range of different data formats, which can also be customised by the user. Creating a built-in data reader that can handle all these formats is impossible. You'll therefore have to write a description of your data format, which `latools` can understand.

The complexity of this data format description will depend on the format of your data, and can vary from a simple 3-4 line snippet, to a baffling array of heiroglyphics. We appreciate that this may be something of a barrier to the beginner.

To make this process as painless as possible, we've put together a step-by-step guide on how to approach this is in the *Data Formats* section.

If you get stuck, head on over to the mailing list, and ask for help.

2. Modify/Make a SRM database File

This contains raw compositional values for the SRMs you use in analysis, and is essential for calibrating your data. `latools` comes with GeoRem 'preferred' compositions for NIST610, NIST612 and NIST614 glasses. If you use any other standards, or are unhappy with the GeoRem 'preferred' values, you'll have to create a new SRM database file.

Instructions on how to do this are in *The SRM File* guide. If you're happy with the GeoRem values, and only use NIST610, NIST612 and NIST614, you can skip this step.

3. Configure LAtools

Once you've got a data description and SRM database that you're happy with, you can create a configuration in `latools` and make it the default for your system. Every time you start a new analysis, `latools` will then automatically use your specific data format description and SRM file.

You can set up multiple configurations, and specify them using the `config` parameter when beggining to *`analyse`* a new dataset. This allows `latools` to easily switch between working on data from different instruments, or using different SRM sets.

Instructions on how to set up and organise configurations are in the *Managing Configurations* section.

1.1.8.2 Data Formats

`latools` can be set up to work with pretty much any conceivable text-based data format. To get your data into `latools`, you need to think about two things:

1. File Structure

At present, `latools` is designed for data that is collected so that each text file contains ablations of a single sample or (a set of) standards, with a name corresponding to the identity of the sample. An ideal data structure would look something like this:

```
data/
  STD-1.csv
  Sample-1.csv
  Sample-2.csv
  Sample-3.csv
  STD-2.csv
```

Where each of the .csv files within the 'data/' contains one or more ablations of a single sample, or numerous standards (i.e. STD-1 could contain ablations of three different standards). The names of the .csv files are used to label the data throughout analysis, so should be unique, and meaningful. Standards are recognised by *`analyse()`* by the presence of identifying characters that are present in all standard names, in this case `'STD'`.

When importing the data, you give *`analyse()`* the `data/` folder, and some information about the SRM identifier (`srm_identifier='STD'`) and the file extension (`extension='.csv'`), and it imports all data files in the folder.

Important: If you data are not in this format (e.g. all your data are stored in one long file), you'll need to convert them into this format to use `latools`. You can find Information on how to do this in the *Preprocessing* pages.

2. Data Format

We tried to make the data import mechanism as simple as possible, but because of the diversity and complexity of formats from different instruments, it can still be a bit tricky to understand. The following will hopefully give you everything you need to write your data format description.

Data Format Description : General Principles

The data format description is stored in a plain-text file, in the JSON format. In practice, the format description consists of a number of names entries with corresponding values, which are read and interpreted by `latools`. A generic JSON file might look something like this:

```
{
    'entry_1': 'value',
    'entry_2': ['this', 'is', 'a', 'list'],
    'entry_3': (['a', 'set', 'of'], 'three', 'values')
}
```

Tip: There are a number of characters that are special in the JSON format (e.g. `/ \ "`). If you want to include these characters in the file, you have to 'escape' them (i.e. mark them as special) by preceding them with a `\`. If this sounds too confusing, you can just use an online formatter to make sure all your entries are JSON-safe.

Required Sections

- `meta_regex` contains information on how to read the 'metadata' in the file header. Each entry has the form:

```
{
    "meta_regex": {
        "line": [["metadata_name"], "Regular Expression with a capture␣
↪group."],
    }
}
```

  Don't worry at this point if 'Regular Expression' and 'capture group' mean nothing to you. *We'll get to that later*.

  Replace `line` with an identifier that selects the line in the data file that the regex is applied to. There are two ways to do this.

  What should `"line"` be?:

  - A number in quotations to pick out a line in the file, e.g. `"3"` to extract the fourth line of the file (remember here that python starts counting at zero). This works well if the file header is *always* the same.

  - A word or string of characters that is *always* in the line (i.e. won't change from file to file). For example you could use `"Date:"`, and `latools` will find the first line in the file that contains `Date:` and apply your regular expression to it. This is useful for formats where the header size can vary depending on the analysis.

Tip: The `meta_regex` component of the dataformat description should contain an entry that finds the 'date' of the analysis. This is used to define the time scale of the whole session which background and drift correction depend upon. This should be specified as''{"line": {["date"], "regex_string"}}'' where `regex_string` isolates the analysis date of the file in a capture group, as demonstrated here. If you don't identify a date in the metadata, `latools` will assume all your analyses were done consecutively with no time gaps between them, and in the order of their sample names. This can cause some unexpected behaviour in the analysis...

- `column_id` contains information on where the column names of the data are, and how to interpret them. This requires 4 specific entries, and should look something like:

```
{
    "column_id": {
        "delimiter": "Character that separates column headings, e.g. \t␣
↪(tab) or , (comma)",
        "timecolumn": "Numeric index of time column. Usually zero (the first␣
↪column). Must be an integer, without quotations.",
        "name_row": "The line number that contains the column headings. Must␣
↪be an integer, without quotations",
        "pattern": "A Regular Expression that identifies valid analyte names␣
↪in a capture group."
    }
}
```

- `genfromtext_args` contains information on how to read the actual data table. `latools` uses Numpy's `genfromtxt()` function to read the raw data, so this section can contain any valid arguments for the `genfromtxt()` function. For example, you might include:

```
{
    "genfromtext_args": {
        "delimiter": "Character that separates data values in rows, e.g. \t␣
↪(tab) or , (comma)",
        "skip_header": "Integer, without quotations, that specifies the␣
↪number of lines at the start of the file that *don't* contain data values.
↪",
    }
}
```

Optional Sections

- `preformat_replace`. Particularly awkward data formats may require some 'cleaning' before they're readable by `genfromtxt()` (e.g. the removal of non-numeric characters). You can do this by optionally including a `preformat_replace` section in your dataformat description. This consists of `{"regex_expression": "replacement_text"}` pairs, which are applied to the data before import. For example:

```
{
    "preformat_replace": {
```

(continues on next page)

(continued from previous page)

```
        "[^0-9, .]+": ""
    }
}
```

will replace all non-numeric characters that are not `.`, `,` or a space with `""` (i.e. no text - remove them). The use of `preformat_replace` should not be necessary for most dataformats. - `time_format`. `latools` attempts to automatically read the `date` information identified by `meta_regex` (using `dateutil`'s `parse()`), but in rare cases this will fail. If it fails, you'll need to manually specify the date format. Specify the date format using standard notation for formatting and reading times. For example:

```
{
    "time_format": "%d-%b-%Y %H:%M:%S"
}
```

will correctly read a time format of "01-Mar-2016 15:23:03".

Regular Expressions (RegEx)

Data import in `latools` makes use of Regular Expressions to identify different parts of your data. Regular expressions are a way of defining *patterns* that allow the computer to extract information from text that isn't exactly the same in every instance. A very basic example, if you apply the pattern:

```
"He's not the Mesiah, (.*)"
```

to `"He's not the Mesiah, he's a very naughty boy!"`, the expression will *match* the text, and you'll get `"he's a very naughty boy!"` in a *capture group*. To break the expression down a bit:

- "He's not the Mesiah, " tells the computer that you're looking for text containing this phrase.
- `.` signifies 'any character'
- `*` signifies 'anywhere between zero and infinity occurrences of `.`
- `()` identifies the 'capture group'. The expression would still match without this, but you wouldn't be able to isolate the text within the capture group afterwards.

What would the capture group get if you applied the expression to `He's not the Mesiah, he just thinks he is...`?

Applying this to metadata extraction, imagine you have a line in your file header like:

```
Acquired      : Oct 29 2015  03:11:05 pm using AcqMethod OB102915.m
```

And you need to extract the date (`Oct 29 2015 03:11:05 pm`). You know that the line always starts with `Acquired [varying number of spaces] :`, and ends with `using AcqMethod [some text]`. The expression:

```
Acquired +: (.*) using.*
```

will get the date in its capture group! For a full explanation of how this works, have a look at this breakdown by Regex101 (Note 'Explanation' section in upper right).

Writing your own Regular Expressions can be tricky to get your head around at first. We suggest using the superb Regex101 site to help you design the Regular Expressions in your data format description. Just copy and paste the text you're working with (e.g. line from file header containing the date), play around with the expression until it works as required, and then copy it across to your dataformat file.

Note: If you're stuck on data formats, submit a question to the mailing list and we'll try to help. If you think you've found a serious problem in the software that will prevent you importing your data, file an issue on the GitHub project page, and we'll look into updating the software to fix the problem.

Writing a new Data Format Description : Step-By-Step

Data produced by the UC Davis Agilent 8800 looks like this:

```
C:\Path\To\Data.D
Intensity Vs Time,CPS
Acquired      : Oct 29 2015  03:11:05 pm using AcqMethod OB102915.m
Time [Sec],Mg24,Mg25,Al27,Ca43,Ca44,Mn55,Sr88,Ba137,Ba138
0.367,666.68,25.00,3100.27,300.00,14205.75,7901.80,166.67,37.50,25.00
...
```

This step-by-step guide will go through the process of writing a dataformat description from scratch for the file.

Tip: We're working from scratch here for illustrative purposes. When doing this in reality, you might find the *get_dataformat_template()* (accessible via `latools.config.get_dataformat_template()`), which creates an annotated data format file for you to adapt.

1. Create an empty file, name it, and give it a `.json` extension. Open the file in your favourite text editor. Data in `.json` files can be stored in lists (comma separated values inside square brackets, e.g. [1,2,3]) or as {'key': 'value'} pairs inside curly brackets.
2. The data format description contains three named sections - `meta_regex`, `column_id` and `genfromtext_args`, which we'll store as {'key': 'value'} pairs. Create empty entries for these in your new `.json` file. Your file should now look like this:

```
{
    "meta_regex": {},
    "column_id": {},
    "genfromtext_args": {}
}
```

3. Define the start time of the analysis. In this case, it's `Oct 29 2015 03:11:05 pm`, but it will be different in other files. We therefore use a regular expression' to define a *pattern* that describes the date. To do this, we'll isolate the line containing the date (line 2 - numbers start at ero in Python!), and head on over to Regex101 to write our expression. Add this expression to the meta_regex ession, with the line number as its key:

```
{
    "meta_regex": {
        "2": [["date"],
               "([A-Z][a-z]+ [0-9]+ [0-9]{4}[ ]+[0-9:]+ [amp]+)"]
    },
    "column_id": {},
    "genfromtext_args": {}
}
```

Tip: Having trouble with Regular Expressions? We really recommend Regex101!

4. Set some parameters that define where the column names are. `name_row` defines which row the column names are in (*3*), `delimeter` describes hat character separates the column names (,), `timecolumn` is the numberical index of the column containing the 'time' data (in this case, *0*). his will grab everything in row 3 that's separated by a comma, and tell `latools` that the first column contains the time info. Now we need to tell t which columns contain the analyte names. We'll do this with a regular expression again, copying the entire column over to Regex101 to help us write he expression. Put all this information into the "column_id" section:

```
{
    "meta_regex": {
        "2": [["date"],
               "([A-Z][a-z]+ [0-9]+ [0-9]{4}[ ]+[0-9:]+ [amp]+)"]
    },
    "column_id": {
        "name_row": 3,
        "delimiter": ",",
        "timecolumn": 0,
        "pattern": "([A-z]{1,2}[0-9]{1,3})"
    },
    "genfromtext_args": {}
}
```

5. Finally, we need to add some parameters that tell `latools` how to read the actual data table. In this case, we want to skip the first 4 lines, nd then tell it that the values are separated by commas. Add this information to the `genfromtext_args` section:

```
{
    "meta_regex": {
        "2": [["date"],
               "([A-Z][a-z]+ [0-9]+ [0-9]{4}[ ]+[0-9:]+ [amp]+)"]
    },
    "column_id": {
        "name_row": 3,
        "delimiter": ",",
        "timecolumn": 0,
        "pattern": "([A-z]{1,2}[0-9]{1,3})"
    },
    "genfromtext_args": {
        "delimiter": ",",
        "skip_header": 4
    }
}
```

6. Test the format description, using the *`test_dataformat()`* function. In Python:

```
import latools as la

my_dataformat = 'path/to/my/dataformat.json'
my_datafile = 'path/to/my/datafile.csv

la.config.test_dataformat(my_datafile, my_dataformat)
```

This will go through the data import process for you file, printing out the results of each stage, so if it fails you can see *where* if failed, and ix the problem.

7. Fix any errors, and you're done! You have a working data description.

I've written my dataformat, now what?

Once you're happy with your data format description, put it in a text file, and save it as 'my_dataformat.json' (obviously replace my_dataformat with something meaningful...). When you want to import data using your newly defined format, you can point `latools` towards it by specifying `dataformat='my_dataformat.dict'` when starting a data analysis. Alternatively, you can define a new *Managing Configurations*, to make this the default data format for your setup.

1.1.8.3 The SRM File

The SRM file contains compositional data for standards. To calibrate raw data standards measured during analysis must be in this database.

File Location

The default SRM table is stored in the *resources* directory within the `latools` install location.

If you wish to use a different SRM table, the path to the new table must be specified in the configuration file or on a case-by-case basis when calibrating your data.

File Format

The SRM file must be stores as a `.csv` file (comma separated values). The full default table has the following columns:

Item	SRM	Value	Uncertainty	Uncertainty_Type	Unit	GeoReM_bibcode	Reference	M	g/g	g/g_err	mol/g	mol/g_err
Se	NIST610	38.0	42.0	95%CL	ug/g	GeoReM 5211	Jochum et al 2011	78.96	0.000138	8.2e-05	1.747e-06	5.319e-07

For completeness, the full SRM file contains a lot of info. You don't need to complete *all* the columns for a new SRM.

Essential Data

The *essential* columns that must be included for `latools` to use a new SRM are:

Item	SRM	mol/g	mol/g_err
Se	NIST610	1.747e-06	5.319e-07

Other columns may be left blank, although we recommend at least adding a note as to where the values come from in the `Reference` column.

Creating/Modifying an SRM File

To create a new table you can either start from scratch (not recommended), or modify a copy of the existing SRM table (recommended).

To get a copy of the existing SRM table, in Python:

```
import latools as la

la.config.copy_SRM_file('path/to/save/location', config='DEFAULT')
```

This will create a copy of the default SRM table, and save it to the specified location. You can then modify the copy as necessary.

To use your new SRM database, you can either specify it manually at the start of a new analysis:

```
import latools as la

eg = la.analyse('data/', srm_file='path/to/srmfile.csv')
```

Or *specify it as part of a configuration*, so that `latools` knows where it is automatically.

1.1.8.4 Managing Configurations

A 'configuration' is how `latools` stores the location of a data format description and SRM file to be used during data import and analysis. In labs working with a single LA-ICPMS system, you can set a default configuration, and then leave this alone. If you're running multiple LA-ICPMS systems, or work with different data formats, you can specify multiple configurations, and specify which one you want to use at the start of analysis, like this:

```
import latools as la

eg = la.analyse('data', config='MY-CONFIG-NAME')
```

Viewing Existing Configurations

You can see a list of currently defined configurations at any time:

```
import latools as la

la.config.print_all()

    Currently defined LAtools configurations:

    REPRODUCE [DO NOT ALTER]
    dataformat: /latools/install/location/resources/data_formats/repro_dataformat.json
    srmfile: /latools/install/location/resources/SRM_GeoRem_Preferred_170622.csv

    UCD-AGILENT [DEFAULT]
    dataformat: /latools/install/location/resources/data_formats/UCD_dataformat.json
    srmfile: /latools/install/location/resources/SRM_GeoRem_Preferred_170622.csv
```

Note how each configuration has a `dataformat` and `srmfile` specified. The `REPRODUCE` configuration is a special case, and should not be modified. All other configurations are listed by name, and the default configuration is marked (in this case there's only one, and it's the default). If you *don't* specify a configuration when you start an analysis, it will use the default one.

Creating a Configuration

Once you've created your own *dataformat description* and/or *SRM File*, you can set up a configuration to use them:

```
import latools as la

# create new config
la.config.create('MY-FANCY-CONFIGURATION',
                 srmfile='path/to/srmfile.csv',
                 dataformat='path/to/dataformat.json',
                 base_on='DEFAULT', make_default=False)

# check it's there
la.config.print_all()

    Currently defined LAtools configurations:

    REPRODUCE [DO NOT ALTER]
    dataformat: /latools/install/location/resources/data_formats/repro_dataformat.json
    srmfile: /latools/install/location/resources/SRM_GeoRem_Preferred_170622.csv

    UCD-AGILENT [DEFAULT]
    dataformat: /latools/install/location/resources/data_formats/UCD_dataformat.json
    srmfile: /latools/install/location/resources/SRM_GeoRem_Preferred_170622.csv

    MY-FANCY-CONFIGURATION
    dataformat: path/to/dataformat.json
    srmfile: path/to/srmfile.csv
```

You should see the new configuration in the list, and unless you specified `make_default=True`, the default should not have changed. The `base_on` argument tells `latools` which existing configuration the new one is based on. This only matters if you're only specifying one of `srmfile` or `dataformat` - whichever you *don't* specify is copied from the `base_on` configuration.

Important: When making a configuration, make sure you store the dataformat and srm files somewhere permanent - if you move or rename these files, the configuration will stop working.

Modifying a Configuration

Once created, configurations can be modified...

```
import latools as la

# modify configuration
la.config.update('MY-FANCY-CONFIGURATION', 'srmfile', 'correct/path/to/srmfile.csv')

    Are you sure you want to change the srmfile parameter of the MY-FANCY-
↪CONFIGURATION configuration?
    It will be changed from:
        path/to/srmfile.csv
    to:
        correct/path/to/srmfile.csv
    > [N/y]: y
    Configuration updated!

# check it's updated
la.config.print_all()
```

(continues on next page)

(continued from previous page)

```
    Currently defined LAtools configurations:

    REPRODUCE [DO NOT ALTER]
    dataformat: /latools/install/location/resources/data_formats/repro_dataformat.json
    srmfile: /latools/install/location/resources/SRM_GeoRem_Preferred_170622.csv

    UCD-AGILENT [DEFAULT]
    dataformat: /latools/install/location/resources/data_formats/UCD_dataformat.json
    srmfile: /latools/install/location/resources/SRM_GeoRem_Preferred_170622.csv

    MY-FANCY-CONFIGURATION
    dataformat: path/to/dataformat.json
    srmfile: correct/path/to/srmfile.csv
```

Deleting a Configuration

Or deleted…

```
import latools as la

# delete configuration
la.config.delete('MY-FANCY-CONFIGURATION')

    Are you sure you want to delete the MY-FANCY-CONFIGURATION configuration?
    > [N/y]: y
    Configuration deleted!

# check it's gone
la.config.print_all()

    Currently defined LAtools configurations:

    REPRODUCE [DO NOT ALTER]
    dataformat: /latools/install/location/resources/data_formats/repro_dataformat.json
    srmfile: /latools/install/location/resources/SRM_GeoRem_Preferred_170622.csv

    UCD-AGILENT [DEFAULT]
    dataformat: /latools/install/location/resources/data_formats/UCD_dataformat.json
    srmfile: /latools/install/location/resources/SRM_GeoRem_Preferred_170622.csv
```

CHAPTER 2

Function Documentation

2.1 LAtools Documentation

2.1.1 latools.analyse object

Main functions for interacting with LAtools.

(c) Oscar Branson : https://github.com/oscarbranson

class latools.latools.**analyse**(*data_folder*, *errorhunt=False*, *config='DEFAULT'*, *dataformat=None*, *extension='.csv'*, *srm_identifier='STD'*, *cmap=None*, *time_format=None*, *internal_standard='Ca43'*, *names='file_names'*, *srm_file=None*, *pbar=None*)

Bases: object

For processing and analysing whole LA - ICPMS datasets.

Parameters

- **data_folder** (*str*) – The path to a directory containing multiple data files.
- **errorhunt** (*bool*) – If True, latools prints the name of each file before it imports the data. This is useful for working out which data file is causing the import to fail.
- **config** (*str*) – The name of the configuration to use for the analysis. This determines which configuration set from the latools.cfg file is used, and overrides the default configuration setup. You might sepcify this if your lab routinely uses two different instruments.
- **dataformat** (*str or dict*) – Either a path to a data format file, or a dataformat dict. See documentation for more details.
- **extension** (*str*) – The file extension of your data files. Defaults to '.csv'.
- **srm_identifier** (*str*) – A string used to separate samples and standards. srm_identifier must be present in all standard measurements. Defaults to 'STD'.
- **cmap** (*dict*) – A dictionary of {analyte: colour} pairs. Colour can be any valid matplotlib colour string, RGB or RGBA sequence, or hex string.

- **time_format** (*str*) – A regex string identifying the time format, used by pandas when created a universal time scale. If unspecified (None), pandas attempts to infer the time format, but in some cases this might not work.
- **internal_standard** (*str*) – The name of the analyte used as an internal standard throughout analysis.
- **names** (*str*) –
  - 'file_names' : use the file names as labels (default)
  - 'metadata_names' : used the 'names' attribute of metadata as the name anything else : use numbers.

folder
Path to the directory containing the data files, as specified by *data_folder*.

Type str

dirname
The name of the directory containing the data files, without the entire path.

Type str

files
A list of all files in *folder*.

Type array_like

param_dir
The directory where parameters are stored.

Type str

report_dir
The directory where plots are saved.

Type str

data
A dict of *latools.D* data objects, labelled by sample name.

Type dict

samples
A list of samples.

Type array_like

analytes
A list of analytes measured.

Type array_like

stds
A list of the *latools.D* objects containing hte SRM data. These must contain srm_identifier in the file name.

Type array_like

srm_identifier
A string present in the file names of all standards.

Type str

cmaps
An analyte - specific colour map, used for plotting.

Type dict

ablation_times(*samples=None*, *subset=None*)

analytes_sorted(*a=None*, *check_ratios=True*)

apply_classifier(*name*, *samples=None*, *subset=None*)
Apply a clustering classifier based on all samples, or a subset.

Parameters

- **name** (*str*) – The name of the classifier to apply.
- **subset** (*str*) – The subset of samples to apply the classifier to.

Returns name

Return type str

autorange(*analyte='total_counts', gwin=5, swin=3, win=20, on_mult=[1.0, 1.5], off_mult=[1.5, 1], transform='log', ploterrs=True, focus_stage='despiked'*)
Automatically separates signal and background data regions.

Automatically detect signal and background regions in the laser data, based on the behaviour of a single analyte. The analyte used should be abundant and homogenous in the sample.

Step 1: Thresholding. The background signal is determined using a gaussian kernel density estimator (kde) of all the data. Under normal circumstances, this kde should find two distinct data distributions, corresponding to 'signal' and 'background'. The minima between these two distributions is taken as a rough threshold to identify signal and background regions. Any point where the trace crosses this thrshold is identified as a 'transition'.

Step 2: Transition Removal. The width of the transition regions between signal and background are then determined, and the transitions are excluded from analysis. The width of the transitions is determined by fitting a gaussian to the smoothed first derivative of the analyte trace, and determining its width at a point where the gaussian intensity is at at *conf* time the gaussian maximum. These gaussians are fit to subsets of the data centered around the transitions regions determined in Step 1, +/- *win* data points. The peak is further isolated by finding the minima and maxima of a second derivative within this window, and the gaussian is fit to the isolated peak.

Parameters

- **analyte** (*str*) – The analyte that autorange should consider. For best results, choose an analyte that is present homogeneously in high concentrations. This can also be 'total_counts' to use the sum of all analytes.
- **gwin** (*int*) – The smoothing window used for calculating the first derivative. Must be odd.
- **win** (*int*) – Determines the width (c +/- win) of the transition data subsets.
- **smwin** (*int*) – The smoothing window used for calculating the second derivative. Must be odd.
- **conf** (*float*) – The proportional intensity of the fitted gaussian tails that determines the transition width cutoff (lower = wider transition regions excluded).
- **trans_mult** (*array_like, len=2*) – Multiples of the peak FWHM to add to the transition cutoffs, e.g. if the transitions consistently leave some bad data proceeding the transition, set trans_mult to [0, 0.5] to ad 0.5 * the FWHM to the right hand side of the limit.
- **focus_stage** (*str*) – Which stage of analysis to apply processing to. Defaults to 'despiked', or rawdata' if not despiked. Can be one of: * 'rawdata': raw data, loaded

from csv file. * 'despiked': despiked data. * 'signal'/'background': isolated signal and background data.

Created by self.separate, after signal and background regions have been identified by self.autorange.

- 'bkgsub': background subtracted data, created by self.bkg_correct
- 'ratios': element ratio data, created by self.ratio.
- 'calibrated': ratio data calibrated to standards, created by self.calibrate.

Returns

- *Outputs added as instance attributes. Returns None.*
- **bkg, sig, trn** (*iterable, bool*) – Boolean arrays identifying background, signal and transision regions
- **bkgrng, sigrng and trnrng** (*iterable*) – (min, max) pairs identifying the boundaries of contiguous True regions in the boolean arrays.

basic_processing(*noise_despiker=True, despike_win=3, despike_nlim=12.0, despike_maxiter=4, autorange_analyte='total_counts', autorange_gwin=5, autorange_swin=3, autorange_win=20, autorange_on_mult=[1.0, 1.5], autorange_off_mult=[1.5, 1], autorange_transform='log', bkg_weight_fwhm=300.0, bkg_n_min=20, bkg_n_max=None, bkg_cstep=None, bkg_filter=False, bkg_f_win=7, bkg_f_n_lim=3, bkg_errtype='stderr', calib_drift_correct=True, calib_srms_used=['NIST610', 'NIST612', 'NIST614'], calib_zero_intercept=True, calib_n_min=10, plots=True*)

bkg_calc_interp1d(*analytes=None, kind=1, n_min=10, n_max=None, cstep=30, bkg_filter=False, f_win=7, f_n_lim=3, errtype='stderr', focus_stage='despiked'*)

Background calculation using a 1D interpolation.

scipy.interpolate.interp1D is used for interpolation.

Parameters

- **analytes** (*str or iterable*) – Which analyte or analytes to calculate.
- **kind** (*str or int*) – Integer specifying the order of the spline interpolation used, or string specifying a type of interpolation. Passed to *scipy.interpolate.interp1D*.
- **n_min** (*int*) – Background regions with fewer than n_min points will not be included in the fit.
- **cstep** (*float or None*) – The interval between calculated background points.
- **filter** (*bool*) – If true, apply a rolling filter to the isolated background regions to exclude regions with anomalously high values. If True, two parameters alter the filter's behaviour:
- **f_win** (*int*) – The size of the rolling window
- **f_n_lim** (*float*) – The number of standard deviations above the rolling mean to set the threshold.
- **focus_stage** (*str*) – Which stage of analysis to apply processing to. Defaults to 'despiked' if present, or 'rawdata' if not. Can be one of: * 'rawdata': raw data, loaded

from csv file. * 'despiked': despiked data. * 'signal'/'background': isolated signal and background data.

Created by self.separate, after signal and background regions have been identified by self.autorange.

- 'bkgsub': background subtracted data, created by self.bkg_correct
- 'ratios': element ratio data, created by self.ratio.
- 'calibrated': ratio data calibrated to standards, created by self.calibrate.

bkg_calc_weightedmean(*analytes=None*, *weight_fwhm=600*, *n_min=20*, *n_max=None*, *cstep=None*, *errtype='stderr'*, *bkg_filter=False*, *f_win=7*, *f_n_lim=3*, *focus_stage='despiked'*)

Background calculation using a gaussian weighted mean.

Parameters

- **analytes** (*str or iterable*) – Which analyte or analytes to calculate.
- **weight_fwhm** (*float*) – The full-width-at-half-maximum of the gaussian used to calculate the weighted average.
- **n_min** (*int*) – Background regions with fewer than n_min points will not be included in the fit.
- **cstep** (*float or None*) – The interval between calculated background points.
- **filter** (*bool*) – If true, apply a rolling filter to the isolated background regions to exclude regions with anomalously high values. If True, two parameters alter the filter's behaviour:
- **f_win** (*int*) – The size of the rolling window
- **f_n_lim** (*float*) – The number of standard deviations above the rolling mean to set the threshold.
- **focus_stage** (*str*) – Which stage of analysis to apply processing to. Defaults to 'despiked' if present, or 'rawdata' if not. Can be one of: * 'rawdata': raw data, loaded from csv file. * 'despiked': despiked data. * 'signal'/'background': isolated signal and background data.

  Created by self.separate, after signal and background regions have been identified by self.autorange.

  - 'bkgsub': background subtracted data, created by self.bkg_correct
  - 'ratios': element ratio data, created by self.ratio.
  - 'calibrated': ratio data calibrated to standards, created by self.calibrate.

bkg_plot(*analytes=None*, *figsize=None*, *yscale='log'*, *ylim=None*, *err='stderr'*, *save=True*)

Plot the calculated background.

Parameters

- **analytes** (*str or iterable*) – Which analyte(s) to plot.
- **figsize** (*tuple*) – The (width, height) of the figure, in inches. If None, calculated based on number of samples.
- **yscale** (*str*) – 'log' (default) or 'linear'.

- **ylim** (*tuple*) – Manually specify the y scale.
- **err** (*str*) – What type of error to plot. Default is stderr.
- **save** (*bool*) – If True, figure is saved.

Returns fig, ax

Return type matplotlib.figure, matplotlib.axes

bkg_subtract(*analytes=None*, *errtype='stderr'*, *focus_stage='despiked'*)

Subtract calculated background from data.

Must run bkg_calc first!

Parameters

- **analytes** (*str or iterable*) – Which analyte(s) to subtract.
- **errtype** (*str*) – Which type of error to propagate. default is 'stderr'.
- **focus_stage** (*str*) – Which stage of analysis to apply processing to. Defaults to 'despiked' if present, or 'rawdata' if not. Can be one of: * 'rawdata': raw data, loaded from csv file. * 'despiked': despiked data. * 'signal'/'background': isolated signal and background data.

  Created by self.separate, after signal and background regions have been identified by self.autorange.

  - 'bkgsub': background subtracted data, created by self.bkg_correct
  - 'ratios': element ratio data, created by self.ratio.
  - 'calibrated': ratio data calibrated to standards, created by self.calibrate.

calculate_mass_fraction(*internal_standard_conc=None*, *analytes=None*, *analyte_masses=None*)

Convert calibrated molar ratios to mass fraction.

Parameters

- **internal_standard_conc** (*float, pandas.DataFrame or str*) – The concentration of the internal standard in your samples. If a string, should be the file name pointing towards the [completed] output of get_sample_list().
- **analytes** (*str of array_like*) – The analytes you want to calculate.
- **analyte_masses** (*dict*) – A dict containing the masses to use for each analyte. If None and the analyte names contain a number, that number is used as the mass. If None and the analyte names do *not* contain a number, the average mass for the element is used.

calibrate(*analytes=None, drift_correct=True, srms_used=['NIST610', 'NIST612', 'NIST614'], zero_intercept=True, n_min=10, reload_srm_database=False*)

Calibrates the data to measured SRM values.

Assumes that y intercept is zero.

Parameters

- **analytes** (*str or iterable*) – Which analytes you'd like to calibrate. Defaults to all.
- **drift_correct** (*bool*) – Whether to pool all SRM measurements into a single calibration, or vary the calibration through the run, interpolating coefficients between measured SRMs.

- **srms_used** (*str or iterable*) – Which SRMs have been measured. Must match names given in SRM data file *exactly*.
- **n_min** (*int*) – The minimum number of data points an SRM measurement must have to be included.

Returns

Return type None

calibration_plot(*analyte_ratios=None*, *datarange=True*, *loglog=False*, *ncol=3*, *srm_group=None*, *percentile_data_cutoff=85*, *save=True*)

clear_calibration()

clear_subsets()
Clears all subsets

correct_spectral_interference(*target_analyte*, *source_analyte*, *f*)
Correct spectral interference.

Subtract interference counts from target_analyte, based on the intensity of a source_analayte and a known fractional contribution (f).

Correction takes the form: target_analyte -= source_analyte * f

Only operates on background-corrected data ('bkgsub'). To undo a correction, rerun *self.bkg_subtract()*.

Example

To correct 44Ca+ for an 88Sr++ interference, where both 43.5 and 44 Da peaks are known: f = abundance(88Sr) / (abundance(87Sr)

counts(44Ca) = counts(44 Da) - counts(43.5 Da) * f

Parameters

- **target_analyte** (*str*) – The name of the analyte to modify.
- **source_analyte** (*str*) – The name of the analyte to base the correction on.
- **f** (*float*) – The fraction of the intensity of the source_analyte to subtract from the target_analyte. Correction is: target_analyte - source_analyte * f

Returns

Return type None

correlation_plots(*x_analyte*, *y_analyte*, *window=15*, *filt=True*, *recalc=False*, *samples=None*, *subset=None*, *outdir=None*)
Plot the local correlation between two analytes.

Parameters

- **y_analyte** (*x_analyte,*) – The names of the x and y analytes to correlate.
- **window** (*int, None*) – The rolling window used when calculating the correlation.
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.
- **recalc** (*bool*) – If True, the correlation is re-calculated, even if it is already present.

Returns

Return type None

crossplot(*analytes=None*, *lognorm=True*, *bins=25*, *filt=False*, *samples=None*, *subset=None*, *figsize=(12*, *12)*, *save=False*, *colourful=True*, *mode='hist2d'*, ***kwargs*)
Plot analytes against each other.

Parameters

- **analytes** (*optional, array_like or str*) – The analyte(s) to plot. Defaults to all analytes.
- **lognorm** (*bool*) – Whether or not to log normalise the colour scale of the 2D histogram.
- **bins** (*int*) – The number of bins in the 2D histogram.
- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.
- **figsize** (*tuple*) – Figure size (width, height) in inches.
- **save** (*bool or str*) – If True, plot is saves as 'crossplot.png', if str plot is saves as str.
- **colourful** (*bool*) – Whether or not the plot should be colourful :).
- **mode** (*str*) – 'hist2d' (default) or 'scatter'

Returns

Return type (fig, axes)

crossplot_filters(*filter_string*, *analytes=None*, *samples=None*, *subset=None*, *filt=None*)
Plot the results of a group of filters in a crossplot.

Parameters

- **filter_string** (*str*) – A string that identifies a group of filters. e.g. 'test' would plot all filters with 'test' in the name.
- **analytes** (*optional, array_like or str*) – The analyte(s) to plot. Defaults to all analytes.

Returns

Return type fig, axes objects

despike(*expdecay_despiker=False*, *exponent=None*, *noise_despiker=True*, *win=3*, *nlim=12.0*, *exponentplot=False*, *maxiter=4*, *autorange_kwargs={}*, *focus_stage='rawdata'*)
Despikes data with exponential decay and noise filters.

Parameters

- **expdecay_despiker** (*bool*) – Whether or not to apply the exponential decay filter.
- **exponent** (*None or float*) – The exponent for the exponential decay filter. If None, it is determined automatically using *find_expocoef*.
- **tstep** (*None or float*) – The timeinterval between measurements. If None, it is determined automatically from the Time variable.
- **noise_despiker** (*bool*) – Whether or not to apply the standard deviation spike filter.
- **win** (*int*) – The rolling window over which the spike filter calculates the trace statistics.
- **nlim** (*float*) – The number of standard deviations above the rolling mean that data are excluded.

- **exponentplot** (*bool*) – Whether or not to show a plot of the automatically determined exponential decay exponent.
- **maxiter** (*int*) – The max number of times that the fitler is applied.
- **focus_stage** (*str*) – Which stage of analysis to apply processing to. Defaults to 'rawdata'. Can be one of: * 'rawdata': raw data, loaded from csv file. * 'despiked': despiked data. * 'signal'/'background': isolated signal and background data.

  Created by self.separate, after signal and background regions have been identified by self.autorange.

  – 'bkgsub': background subtracted data, created by self.bkg_correct

  – 'ratios': element ratio data, created by self.ratio.

  – 'calibrated': ratio data calibrated to standards, created by self.calibrate.

Returns

Return type None

export_traces(*outdir=None*, *focus_stage=None*, *analytes=None*, *samples=None*, *subset='All_Analyses'*, *filt=False*, *zip_archive=False*)

Function to export raw data.

Parameters

- **outdir** (*str*) – directory to save toe traces. Defaults to 'main-dir-name_export'.
- **focus_stage** (*str*) – The name of the analysis stage to export.
  – 'rawdata': raw data, loaded from csv file.
  – 'despiked': despiked data.
  – 'signal'/'background': isolated signal and background data. Created by self.separate, after signal and background regions have been identified by self.autorange.
  – 'bkgsub': background subtracted data, created by self.bkg_correct
  – 'ratios': element ratio data, created by self.ratio.
  – 'calibrated': ratio data calibrated to standards, created by self.calibrate.

  Defaults to the most recent stage of analysis.
- **analytes** (*str or array_like*) – Either a single analyte, or list of analytes to export. Defaults to all analytes.
- **samples** (*str or array_like*) – Either a single sample name, or list of samples to export. Defaults to all samples.
- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.

filter_clear(*samples=None*, *subset=None*)

Clears (deletes) all data filters.

filter_clustering(*analytes*, *filt=False*, *normalise=True*, *method='kmeans'*, *include_time=False*, *samples=None*, *sort=True*, *subset=None*, *level='sample'*, *min_data=10*, ***kwargs*)

Applies an n - dimensional clustering filter to the data.

Parameters

- **analytes** (*str*) – The analyte(s) that the filter applies to.
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.
- **normalise** (*bool*) – Whether or not to normalise the data to zero mean and unit variance. Reccomended if clustering based on more than 1 analyte. Uses *sklearn.preprocessing.scale*.
- **method** (*str*) – Which clustering algorithm to use:
  - 'meanshift': The *sklearn.cluster.MeanShift* algorithm. Automatically determines number of clusters in data based on the *bandwidth* of expected variation.
  - 'kmeans': The *sklearn.cluster.KMeans* algorithm. Determines the characteristics of a known number of clusters within the data. Must provide *n_clusters* to specify the expected number of clusters.
- **level** (*str*) – Whether to conduct the clustering analysis at the 'sample' or 'population' level.
- **include_time** (*bool*) – Whether or not to include the Time variable in the clustering analysis. Useful if you're looking for spatially continuous clusters in your data, i.e. this will identify each spot in your analysis as an individual cluster.
- **samples** (*optional, array_like or None*) – Which samples to apply this filter to. If None, applies to all samples.
- **sort** (*bool*) – Whether or not you want the cluster labels to be sorted by the mean magnitude of the signals they are based on (0 = lowest)
- **min_data** (*int*) – The minimum number of data points that should be considered by the filter. Default = 10.
- ****kwargs** – Parameters passed to the clustering algorithm specified by *method*.
- **Parameters** (*K-Means*) –

  bandwidth [str or float] The bandwith (float) or bandwidth method ('scott' or 'silverman') used to estimate the data bandwidth.

  bin_seeding [bool] Modifies the behaviour of the meanshift algorithm. Refer to sklearn.cluster.meanshift documentation.
- **Parameters** –

  n_clusters [int] The number of clusters expected in the data.

Returns

Return type None

filter_correlation(*x_analyte*, *y_analyte*, *window=None*, *r_threshold=0.9*, *p_threshold=0.05*, *filt=True*, *samples=None*, *subset=None*)

Applies a correlation filter to the data.

Calculates a rolling correlation between every *window* points of two analytes, and excludes data where their Pearson's R value is above *r_threshold* and statistically significant.

Data will be excluded where their absolute R value is greater than *r_threshold* AND the p - value associated with the correlation is less than *p_threshold*. i.e. only correlations that are statistically significant are considered.

Parameters

- **y_analyte** (*x_analyte,*) – The names of the x and y analytes to correlate.

- **window** (*int, None*) – The rolling window used when calculating the correlation.
- **r_threshold** (*float*) – The correlation index above which to exclude data. Note: the absolute pearson R value is considered, so negative correlations below *-r_threshold* will also be excluded.
- **p_threshold** (*float*) – The significant level below which data are excluded.
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.

Returns

Return type None

filter_defragment(*threshold*, *mode='include'*, *filt=True*, *samples=None*, *subset=None*)
Remove 'fragments' from the calculated filter

Parameters

- **threshold** (*int*) – Contiguous data regions that contain this number or fewer points are considered 'fragments'
- **mode** (*str*) – Specifies wither to 'include' or 'exclude' the identified fragments.
- **filt** (*bool or filt string*) – Which filter to apply the defragmenter to. Defaults to True
- **samples** (*array_like or None*) – Which samples to apply this filter to. If None, applies to all samples.
- **subset** (*str or number*) – The subset of samples (defined by make_subset) you want to apply the filter to.

Returns

Return type None

filter_effect(*analytes=None, stats=['mean', 'std'], filt=True*)
Quantify the effects of the active filters.

Parameters

- **analytes** (*str or list*) – Which analytes to consider.
- **stats** (*list*) – Which statistics to calculate.
- **file** (*valid filter string or bool*) – Which filter to consider. If True, applies all active filters.

Returns Contains statistics calculated for filtered and unfiltered data, and the filtered/unfiltered ratio.

Return type pandas.DataFrame

filter_exclude_downhole(*threshold*, *filt=True*, *samples=None*, *subset=None*)
Exclude all points down-hole (after) the first excluded data.

Parameters

- **threhold** (*int*) – The minimum number of contiguous excluded data points that must exist before downhole exclusion occurs.
- **file** (*valid filter string or bool*) – Which filter to consider. If True, applies to currently active filters.

filter_gradient_threshold(*analyte*, *threshold*, *win=15*, *recalc=True*, *win_mode='mid'*, *win_exclude_outside=True*, *absolute_gradient=True*, *samples=None*, *subset=None*)

Calculate a gradient threshold filter to the data.

Generates two filters above and below the threshold value for a given analyte.

Parameters

- **analyte** (*str*) – The analyte that the filter applies to.
- **win** (*int*) – The window over which to calculate the moving gradient
- **threshold** (*float*) – The threshold value.
- **recalc** (*bool*) – Whether or not to re-calculate the gradients.
- **win_mode** (*str*) – Whether the rolling window should be centered on the left, middle or centre of the returned value. Can be 'left', 'mid' or 'right'.
- **win_exclude_outside** (*bool*) – If True, regions at the start and end where the gradient cannot be calculated (depending on win_mode setting) will be excluded by the filter.
- **absolute_gradient** (*bool*) – If True, the filter is applied to the absolute gradient (i.e. always positive), allowing the selection of 'flat' vs 'steep' regions regardless of slope direction. If Falose, the sign of the gradient matters, allowing the selection of positive or negative slopes only.
- **samples** (*array_like or None*) – Which samples to apply this filter to. If None, applies to all samples.
- **subset** (*str or number*) – The subset of samples (defined by make_subset) you want to apply the filter to.

Returns

Return type None

filter_gradient_threshold_percentile(*analyte*, *percentiles*, *level='population'*, *win=15*, *filt=False*, *samples=None*, *subset=None*)

Calculate a gradient threshold filter to the data.

Generates two filters above and below the threshold value for a given analyte.

Parameters

- **analyte** (*str*) – The analyte that the filter applies to.
- **win** (*int*) – The window over which to calculate the moving gradient
- **percentiles** (*float or iterable of len=2*) – The percentile values.
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.
- **samples** (*array_like or None*) – Which samples to apply this filter to. If None, applies to all samples.
- **subset** (*str or number*) – The subset of samples (defined by make_subset) you want to apply the filter to.

Returns

Return type None

filter_nremoved(*filt=True*, *quiet=False*)

Report how many data are removed by the active filters.

filter_off(*filt=None*, *analyte=None*, *samples=None*, *subset=None*, *show_status=False*)
Turns data filters off for particular analytes and samples.

Parameters

- **filt** (*optional, str or array_like*) – Name, partial name or list of names of filters. Supports partial matching. i.e. if 'cluster' is specified, all filters with 'cluster' in the name are activated. Defaults to all filters.
- **analyte** (*optional, str or array_like*) – Name or list of names of analytes. Defaults to all analytes.
- **samples** (*optional, array_like or None*) – Which samples to apply this filter to. If None, applies to all samples.

Returns

Return type None

filter_on(*filt=None*, *analyte=None*, *samples=None*, *subset=None*, *show_status=False*)
Turns data filters on for particular analytes and samples.

Parameters

- **filt** (*optional, str or array_like*) – Name, partial name or list of names of filters. Supports partial matching. i.e. if 'cluster' is specified, all filters with 'cluster' in the name are activated. Defaults to all filters.
- **analyte** (*optional, str or array_like*) – Name or list of names of analytes. Defaults to all analytes.
- **samples** (*optional, array_like or None*) – Which samples to apply this filter to. If None, applies to all samples.

Returns

Return type None

filter_reports(*analytes*, *filt_str='all'*, *nbin=5*, *samples=None*, *outdir=None*, *subset='All_Samples'*)
Plot filter reports for all filters that contain `filt_str` in the name.

filter_status(*sample=None*, *subset=None*, *stds=False*)
Prints the current status of filters for specified samples.

Parameters

- **sample** (*str*) – Which sample to print.
- **subset** (*str*) – Specify a subset
- **stds** (*bool*) – Whether or not to include standards.

filter_threshold(*analyte*, *threshold*, *samples=None*, *subset=None*)
Applies a threshold filter to the data.

Generates two filters above and below the threshold value for a given analyte.

Parameters

- **analyte** (*str*) – The analyte that the filter applies to.
- **threshold** (*float*) – The threshold value.
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.

- **samples** (*array_like or None*) – Which samples to apply this filter to. If None, applies to all samples.
- **subset** (*str or number*) – The subset of samples (defined by make_subset) you want to apply the filter to.

Returns

Return type None

filter_threshold_percentile(*analyte*, *percentiles*, *level='population'*, *filt=False*, *samples=None*, *subset=None*)

Applies a threshold filter to the data.

Generates two filters above and below the threshold value for a given analyte.

Parameters

- **analyte** (*str*) – The analyte that the filter applies to.
- **percentiles** (*float or iterable of len=2*) – The percentile values.
- **level** (*str*) – Whether to calculate percentiles from the entire dataset ('population') or for each individual sample ('individual')
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.
- **samples** (*array_like or None*) – Which samples to apply this filter to. If None, applies to all samples.
- **subset** (*str or number*) – The subset of samples (defined by make_subset) you want to apply the filter to.

Returns

Return type None

filter_trim(*start=1*, *end=1*, *filt=True*, *samples=None*, *subset=None*)

Remove points from the start and end of filter regions.

Parameters

- **end** (*start,*) – The number of points to remove from the start and end of the specified filter.
- **filt** (*valid filter string or bool*) – Which filter to trim. If True, applies to currently active filters.

find_expcoef(*nsd_below=0.0*, *plot=False*, *trimlim=None*, *autorange_kwargs={}*)

Determines exponential decay coefficient for despike filter.

Fits an exponential decay function to the washout phase of standards to determine the washout time of your laser cell. The exponential coefficient reported is *nsd_below* standard deviations below the fitted exponent, to ensure that no real data is removed.

Total counts are used in fitting, rather than a specific analyte.

Parameters

- **nsd_below** (*float*) – The number of standard deviations to subtract from the fitted coefficient when calculating the filter exponent.
- **plot** (*bool or str*) – If True, creates a plot of the fit, if str the plot is to the location specified in str.

- **trimlim** (*float*) – A threshold limit used in determining the start of the exponential decay region of the washout. Defaults to half the increase in signal over background. If the data in the plot don't fall on an exponential decay line, change this number. Normally you'll need to increase it.

Returns

Return type None

fit_classifier(*name*, *analytes*, *method*, *samples=None*, *subset=None*, *filt=True*, *sort_by=0*, ***kwargs*)

Create a clustering classifier based on all samples, or a subset.

Parameters

- **name** (*str*) – The name of the classifier.
- **analytes** (*str or iterable*) – Which analytes the clustering algorithm should consider.
- **method** (*str*) – Which clustering algorithm to use. Can be:

  'meanshift' The *sklearn.cluster.MeanShift* algorithm. Automatically determines number of clusters in data based on the *bandwidth* of expected variation.

  'kmeans' The *sklearn.cluster.KMeans* algorithm. Determines the characteristics of a known number of clusters within the data. Must provide *n_clusters* to specify the expected number of clusters.
- **samples** (*iterable*) – list of samples to consider. Overrides 'subset'.
- **subset** (*str*) – The subset of samples used to fit the classifier. Ignored if 'samples' is specified.
- **sort_by** (*int*) – Which analyte the resulting clusters should be sorted by - defaults to 0, which is the first analyte.
- ****kwargs** – method-specific keyword parameters - see below.
- **Parameters** (*Meanshift*) –

  bandwidth [str or float] The bandwith (float) or bandwidth method ('scott' or 'silverman') used to estimate the data bandwidth.

  bin_seeding [bool] Modifies the behaviour of the meanshift algorithm. Refer to sklearn.cluster.meanshift documentation.
- **- Means Parameters** (*K*) –

  n_clusters [int] The number of clusters expected in the data.

Returns **name**

Return type str

get_background(*n_min=10*, *n_max=None*, *focus_stage='despiked'*, *bkg_filter=False*, *f_win=5*, *f_n_lim=3*)

Extract all background data from all samples on universal time scale. Used by both 'polynomial' and 'weightedmean' methods.

Parameters

- **n_min** (*int*) – The minimum number of points a background region must have to be included in calculation.

- **n_max** (*int*) – The maximum number of points a background region must have to be included in calculation.
- **filter** (*bool*) – If true, apply a rolling filter to the isolated background regions to exclude regions with anomalously high values. If True, two parameters alter the filter's behaviour:
- **f_win** (*int*) – The size of the rolling window
- **f_n_lim** (*float*) – The number of standard deviations above the rolling mean to set the threshold.
- **focus_stage** (*str*) – Which stage of analysis to apply processing to. Defaults to 'despiked' if present, or 'rawdata' if not. Can be one of: * 'rawdata': raw data, loaded from csv file. * 'despiked': despiked data. * 'signal'/'background': isolated signal and background data.

  Created by self.separate, after signal and background regions have been identified by self.autorange.

  - 'bkgsub': background subtracted data, created by self.bkg_correct
  - 'ratios': element ratio data, created by self.ratio.
  - 'calibrated': ratio data calibrated to standards, created by self.calibrate.

Returns

Return type pandas.DataFrame object containing background data.

get_focus(*filt=False*, *samples=None*, *subset=None*, *nominal=False*)
Collect all data from all samples into a single array. Data from standards is not collected.

Parameters

- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.
- **samples** (*str or list*) – which samples to get
- **subset** (*str or int*) – which subset to get

Returns

Return type None

get_gradients(*analytes=None*, *win=15*, *filt=False*, *samples=None*, *subset=None*, *recalc=True*)
Collect all data from all samples into a single array. Data from standards is not collected.

Parameters

- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.
- **samples** (*str or list*) – which samples to get
- **subset** (*str or int*) – which subset to get

Returns

Return type None

get_sample_list(*save_as=None*, *overwrite=False*)
Save a csv list of of all samples to be populated with internal standard concentrations.

Parameters **save_as** (*str*) – Location to save the file. Defaults to the export directory.

getstats(*save=True*, *filename=None*, *samples=None*, *subset=None*, *ablation_time=False*)
Return pandas dataframe of all sample statistics.

gradient_crossplot(*analytes=None*, *win=15*, *lognorm=True*, *bins=25*, *filt=False*, *samples=None*, *subset=None*, *figsize=(12, 12)*, *save=False*, *colourful=True*, *mode='hist2d'*, *recalc=True*, ***kwargs*)
Plot analyte gradients against each other.

Parameters

- **analytes** (*optional, array_like or str*) – The analyte(s) to plot. Defaults to all analytes.
- **lognorm** (*bool*) – Whether or not to log normalise the colour scale of the 2D histogram.
- **bins** (*int*) – The number of bins in the 2D histogram.
- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.
- **figsize** (*tuple*) – Figure size (width, height) in inches.
- **save** (*bool or str*) – If True, plot is saves as 'crossplot.png', if str plot is saves as str.
- **colourful** (*bool*) – Whether or not the plot should be colourful :).
- **mode** (*str*) – 'hist2d' (default) or 'scatter'
- **recalc** (*bool*) – Whether to re-calculate the gradients, or use existing gradients.

Returns

Return type (fig, axes)

gradient_histogram(*analytes=None*, *win=15*, *filt=False*, *bins=None*, *samples=None*, *subset=None*, *recalc=True*, *ncol=4*)
Plot a histogram of the gradients in all samples.

Parameters

- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.
- **bins** (*None or array_like*) – The bins to use in the histogram
- **samples** (*str or list*) – which samples to get
- **subset** (*str or int*) – which subset to get
- **recalc** (*bool*) – Whether to re-calculate the gradients, or use existing gradients.

Returns

Return type fig, ax

gradient_plots(*analytes=None, win=None, samples=None, ranges=False, focus=None, filt=False, recalc=False, outdir=None, figsize=[10, 4], subset='All_Analyses'*)
Plot analyte gradients as a function of time.

Parameters

- **analytes** (*optional, array_like or str*) – The analyte(s) to plot. Defaults to all analytes.
- **samples** (*optional, array_like or str*) – The sample(s) to plot. Defaults to all samples.
- **ranges** (*bool*) – Whether or not to show the signal/backgroudn regions identified by 'autorange'.
- **focus** (*str*) – The focus 'stage' of the analysis to plot. Can be 'rawdata', 'despiked':, 'signal', 'background', 'bkgsub', 'ratios' or 'calibrated'.
- **outdir** (*str*) – Path to a directory where you'd like the plots to be saved. Defaults to 'reports/[focus]' in your data directory.
- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.
- **scale** (*str*) – If 'log', plots the data on a log scale.
- **figsize** (*array_like*) – Array of length 2 specifying figure [width, height] in inches.
- **stats** (*bool*) – Whether or not to overlay the mean and standard deviations for each trace.
- **err** (*stat,*) – The names of the statistic and error components to plot. Deafaults to 'nanmean' and 'nanstd'.

Returns

Return type None

histograms(*analytes=None*, *bins=25*, *logy=False*, *filt=False*, *colourful=True*)

Plot histograms of analytes.

Parameters

- **analytes** (*optional, array_like or str*) – The analyte(s) to plot. Defaults to all analytes.
- **bins** (*int*) – The number of bins in each histogram (default = 25)
- **logy** (*bool*) – If true, y axis is a log scale.
- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.
- **colourful** (*bool*) – If True, histograms are colourful :)

Returns

Return type (fig, axes)

make_subset(*samples=None*, *name=None*)

Creates a subset of samples, which can be treated independently.

Parameters

- **samples** (*str or array_like*) – Name of sample, or list of sample names.
- **name** (*(optional) str or number*) – The name of the sample group. Defaults to n + 1, where n is the highest existing group number

minimal_export(*target_analytes=None*, *path=None*)
Exports a analysis parameters, standard info and a minimal dataset, which can be imported by another user.

Parameters

- **target_analytes** (*str or iterable*) – Which analytes to include in the export. If specified, the export will contain these analytes, and all other analytes used during data processing (e.g. during filtering). If not specified, all analytes are exported.
- **path** (*str*) – Where to save the minimal export. If it ends with .zip, a zip file is created. If it's a folder, all data are exported to a folder.

optimisation_plots(*overlay_alpha=0.5*, *samples=None*, *subset=None*, ***kwargs*)
Plot the result of signal_optimise.

signal_optimiser must be run first, and the output stored in the *opt* attribute of the latools.D object.

Parameters

- **d** (*latools.D object*) – A latools data object.
- **overlay_alpha** (*float*) – The opacity of the threshold overlays. Between 0 and 1.
- ****kwargs** – Passed to *tplot*

optimise_signal(*analytes*, *min_points=5*, *threshold_mode='kde_first_max'*, *threshold_mult=1.0*, *x_bias=0*, *filt=True*, *weights=None*, *mode='minimise'*, *samples=None*, *subset=None*)
Optimise data selection based on specified analytes.

Identifies the longest possible contiguous data region in the signal where the relative standard deviation (std) and concentration of all analytes is minimised.

Optimisation is performed via a grid search of all possible contiguous data regions. For each region, the mean std and mean scaled analyte concentration ('amplitude') are calculated.

The size and position of the optimal data region are identified using threshold std and amplitude values. Thresholds are derived from all calculated stds and amplitudes using the method specified by *threshold_mode*. For example, using the 'kde_max' method, a probability density function (PDF) is calculated for std and amplitude values, and the threshold is set as the maximum of the PDF. These thresholds are then used to identify the size and position of the longest contiguous region where the std is below the threshold, and the amplitude is either below the threshold.

All possible regions of the data that have at least *min_points* are considered.

For a graphical demonstration of the action of signal_optimiser, use *optimisation_plot*.

Parameters

- **d** (*latools.D object*) – An latools data object.
- **analytes** (*str or array_like*) – Which analytes to consider.
- **min_points** (*int*) – The minimum number of contiguous points to consider.
- **threshold_mode** (*str*) – The method used to calculate the optimisation thresholds. Can be 'mean', 'median', 'kde_max' or 'bayes_mvs', or a custom function. If a function, must take a 1D array, and return a single, real number.
- **weights** (*array_like of length len(analytes)*) – An array of numbers specifying the importance of each analyte considered. Larger number makes the analyte have a greater effect on the optimisation. Default is None.

ratio(*internal_standard=None*, *analytes=None*)
Calculates the ratio of all analytes to a single analyte.

Parameters **internal_standard** (*str*) – The name of the analyte to divide all other analytes by.

Returns

Return type None

read_internal_standard_concs(*sample_concs=None*)
Load in a per-sample list of internal sample concentrations.

sample_stats(*analytes=None, filt=True, stats=['mean', 'std'], include_srms=False, eachtrace=True, focus_stage=None, csf_dict={}*)
Calculate sample statistics.

Returns samples, analytes, and arrays of statistics of shape (samples, analytes). Statistics are calculated from the 'focus' data variable, so output depends on how the data have been processed.

Included stat functions:

- `mean()`: arithmetic mean
- `std()`: arithmetic standard deviation
- `se()`: arithmetic standard error
- `H15_mean()`: Huber mean (outlier removal)
- `H15_std()`: Huber standard deviation (outlier removal)
- `H15_se()`: Huber standard error (outlier removal)

Parameters

- **analytes** (*optional, array_like or str*) – The analyte(s) to calculate statistics for. Defaults to all analytes.
- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.
- **stats** (*array_like*) – take a single array_like input, and return a single statistic. list of functions or names (see above) or functions that Function should be able to cope with NaN values.
- **eachtrace** (*bool*) – Whether to calculate the statistics for each analysis spot individually, or to produce per - sample means. Default is True.
- **focus_stage** (*str*) – Which stage of analysis to calculate stats for. Defaults to current stage. Can be one of: * 'rawdata': raw data, loaded from csv file. * 'despiked': despiked data. * 'signal'/'background': isolated signal and background data.

  Created by self.separate, after signal and background regions have been identified by self.autorange.

  – 'bkgsub': background subtracted data, created by self.bkg_correct

  – 'ratios': element ratio data, created by self.ratio.

  – 'calibrated': ratio data calibrated to standards, created by self.calibrate.

  – 'massfrac': mass fraction of each element.

Returns Adds dict to analyse object containing samples, analytes and functions and data.

Return type None

save_log(*directory=None*, *logname=None*, *header=None*)
Save analysis.lalog in specified location

set_focus(*focus_stage=None*, *samples=None*, *subset=None*)
Set the 'focus' attribute of the data file.

The 'focus' attribute of the object points towards data from a particular stage of analysis. It is used to identify the 'working stage' of the data. Processing functions operate on the 'focus' stage, so if steps are done out of sequence, things will break.

Names of analysis stages:

- 'rawdata': raw data, loaded from csv file when object is initialised.
- 'despiked': despiked data.
- 'signal'/'background': isolated signal and background data, padded with np.nan. Created by self.separate, after signal and background regions have been identified by self.autorange.
- 'bkgsub': background subtracted data, created by self.bkg_correct
- 'ratios': element ratio data, created by self.ratio.
- 'calibrated': ratio data calibrated to standards, created by self.calibrate.

Parameters **focus** (*str*) – The name of the analysis stage desired.

Returns

Return type None

srm_build_calib_table()
Combine SRM database values and identified measured values into a calibration database.

srm_compile_measured(*n_min=10*, *focus_stage='ratios'*)
Compile mean and standard errors of measured SRMs

Parameters **n_min** (*int*) – The minimum number of points to consider as a valid measurement. Default = 10.

srm_id_auto(*srms_used=['NIST610', 'NIST612', 'NIST614']*, *analytes=None*, *n_min=10*, *reload_srm_database=False*)
Function for automarically identifying SRMs using KMeans clustering.

KMeans is performed on the log of SRM composition, which aids separation of relatively similar SRMs within a large compositional range.

Parameters

- **srms_used** (*iterable*) – Which SRMs have been used. Must match SRM names in SRM database *exactly* (case sensitive!).
- **analytes** (*array_like*) – Which analyte ratios to base the identification on. If None, all analyte ratios are used (default).
- **n_min** (*int*) – The minimum number of data points a SRM measurement must contain to be included.
- **reload_srm_database** (*bool*) – Whether or not to re-load the SRM database before running the function.

srm_load_database(*srms_used=None*, *reload=False*)

statplot(*analytes=None*, *samples=None*, *figsize=None*, *stat='mean'*, *err='std'*, *subset=None*)
Function for visualising per-ablation and per-sample means.

Parameters

- **analytes** (*str or iterable*) – Which analyte(s) to plot
- **samples** (*str or iterable*) – Which sample(s) to plot
- **figsize** (*tuple*) – Figure (width, height) in inches
- **stat** (*str*) – Which statistic to plot. Must match the name of the functions used in 'sample_stats'.
- **err** (*str*) – Which uncertainty to plot.
- **subset** (*str*) – Which subset of samples to plot.

trace_plots(*analytes=None, samples=None, ranges=False, focus=None, outdir=None, filt=None, scale='log', figsize=[10, 4], stats=False, stat='nanmean', err='nanstd', subset=None*)
Plot analytes as a function of time.

Parameters

- **analytes** (*optional, array_like or str*) – The analyte(s) to plot. Defaults to all analytes.
- **samples** (*optional, array_like or str*) – The sample(s) to plot. Defaults to all samples.
- **ranges** (*bool*) – Whether or not to show the signal/backgroudn regions identified by 'autorange'.
- **focus** (*str*) – The focus 'stage' of the analysis to plot. Can be 'rawdata', 'despiked':, 'signal', 'background', 'bkgsub', 'ratios' or 'calibrated'.
- **outdir** (*str*) – Path to a directory where you'd like the plots to be saved. Defaults to 'reports/[focus]' in your data directory.
- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.
- **scale** (*str*) – If 'log', plots the data on a log scale.
- **figsize** (*array_like*) – Array of length 2 specifying figure [width, height] in inches.
- **stats** (*bool*) – Whether or not to overlay the mean and standard deviations for each trace.
- **err** (*stat,*) – The names of the statistic and error components to plot. Deafaults to 'nanmean' and 'nanstd'.

Returns

Return type None

zeroscreen(*focus_stage=None*)
Remove all points containing data below zero (which are impossible!)

latools.latools.**reproduce**(*past_analysis*, *plotting=False*, *data_folder=None*, *srm_table=None*, *custom_stat_functions=None*)
Reproduce a previous analysis exported with `latools.analyse.minimal_export()`

For normal use, supplying *log_file* and specifying a plotting option should be enough to reproduce an analysis. All requisites (raw data, SRM table and any custom stat functions) will then be imported from the minimal_export folder.

You may also specify your own raw_data, srm_table and custom_stat_functions, if you wish.

Parameters

- **log_file** (*str*) – The path to the log file produced by `minimal_export()`.
- **plotting** (*bool*) – Whether or not to output plots.
- **data_folder** (*str*) – Optional. Specify a different data folder. Data folder should normally be in the same folder as the log file.
- **srm_table** (*str*) – Optional. Specify a different SRM table. SRM table should normally be in the same folder as the log file.
- **custom_stat_functions** (*str*) – Optional. Specify a python file containing custom stat functions for use by reproduce. Any custom stat functions should normally be included in the same folder as the log file.

2.1.2 latools.D object

The Data object, used to store and manipulate the data contained in a single laser ablation files. A core dependency of LAtools.

(c) Oscar Branson : https://github.com/oscarbranson

class latools.D_obj.**D**(*data_file*, *dataformat=None*, *errorhunt=False*, *cmap=None*, *internal_standard=None*, *name='file_names'*)

Bases: object

Container for data from a single laser ablation analysis.

Parameters

- **data_file** (*str*) – The path to a data file.
- **errorhunt** (*bool*) – Whether or not to print each data file name before import. This is useful for tracing which data file is causing the import to fail.
- **dataformat** (*dict*) – A dataformat dict. See documentation for more details.

sample
Sample name.

Type str

meta
Metadata extracted from the csv header. Contents varies, depending on your *dataformat*.

Type dict

analytes
A list of analytes measured.

Type array_like

data
A dictionary containing the raw data, and modified data from each processing stage. Entries can be:

- 'rawdata': created during initialisation.
- 'despiked': created by *despike*

- 'signal': created by *autorange*
- 'background': created by *autorange*
- 'bkgsub': created by *bkg_correct*
- 'ratios': created by *ratio*
- 'calibrated': created by *calibrate*

Type dict

focus
A dictionary containing one item from *data*. This is the currently 'active' data that processing functions will work on. This data is also directly available as class attributes with the same names as the items in *focus*.

Type dict

focus_stage
Identifies which item in *data* is currently assigned to *focus*.

Type str

cmap
A dictionary containing hex colour strings corresponding to each measured analyte.

Type dict

bkg, sig, trn
Boolean arrays identifying signal, background and transition regions. Created by *autorange*.

Type array_like, bool

bkgrng, sigrng, trnrng
An array of shape (n, 2) containing pairs of values that describe the Time limits of background, signal and transition regions.

Type array_like

ns
An integer array the same length as the data, where each analysis spot is labelled with a unique number. Used for separating analysis spots when calculating sample statistics.

Type array_like

filt
An object for storing, selecting and creating data filters.F

Type filt object

ablation_times()
Function for calculating the ablation time for each ablation.

Returns

Return type dict of times for each ablation.

analytes_sorted(*a=None*)

autorange(*analyte='total_counts', gwin=5, swin=3, win=30, on_mult=[1.0, 1.0], off_mult=[1.0, 1.5], ploterrs=True, transform='log', **kwargs*)
Automatically separates signal and background data regions.

Automatically detect signal and background regions in the laser data, based on the behaviour of a single analyte. The analyte used should be abundant and homogenous in the sample.

Step 1: Thresholding. The background signal is determined using a gaussian kernel density estimator (kde) of all the data. Under normal circumstances, this kde should find two distinct data distributions, corresponding to 'signal' and 'background'. The minima between these two distributions is taken as a rough threshold to identify signal and background regions. Any point where the trace crosses this thrshold is identified as a 'transition'.

Step 2: Transition Removal. The width of the transition regions between signal and background are then determined, and the transitions are excluded from analysis. The width of the transitions is determined by fitting a gaussian to the smoothed first derivative of the analyte trace, and determining its width at a point where the gaussian intensity is at at *conf* time the gaussian maximum. These gaussians are fit to subsets of the data centered around the transitions regions determined in Step 1, +/- *win* data points. The peak is further isolated by finding the minima and maxima of a second derivative within this window, and the gaussian is fit to the isolated peak.

Parameters

- **analyte** (*str*) – The analyte that autorange should consider. For best results, choose an analyte that is present homogeneously in high concentrations.
- **gwin** (*int*) – The smoothing window used for calculating the first derivative. Must be odd.
- **win** (*int*) – Determines the width (c +/- win) of the transition data subsets.
- **and off_mult** (*on_mult*) – Factors to control the width of the excluded transition regions. A region n times the full - width - half - maximum of the transition gradient will be removed either side of the transition center. *on_mult* and *off_mult* refer to the laser - on and laser - off transitions, respectively. See manual for full explanation. Defaults to (1.5, 1) and (1, 1.5).

Returns

- *Outputs added as instance attributes. Returns None.*
- **bkg, sig, trn** (*iterable, bool*) – Boolean arrays identifying background, signal and transision regions
- **bkgrng, sigrng and trnrng** (*iterable*) – (min, max) pairs identifying the boundaries of contiguous True regions in the boolean arrays.

autorange_plot(*analyte='total_counts', gwin=7, swin=None, win=20, on_mult=[1.5, 1.0], off_mult=[1.0, 1.5], transform='log'*)
Plot a detailed autorange report for this sample.

bkg_subtract(*analyte*, *bkg*, *ind=None*, *focus_stage='despiked'*)
Subtract provided background from signal (focus stage).

Results is saved in new 'bkgsub' focus stage

Returns

Return type None

calc_correlation(*x_analyte*, *y_analyte*, *window=15*, *filt=True*, *recalc=True*)
Calculate local correlation between two analytes.

Parameters

- **y_analyte** (*x_analyte,*) – The names of the x and y analytes to correlate.
- **window** (*int, None*) – The rolling window used when calculating the correlation.
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.

- **recalc** (*bool*) – If True, the correlation is re-calculated, even if it is already present.

Returns

Return type None

calc_mass_fraction(*internal_standard_conc*, *analytes=None*, *analyte_masses=None*)

calibrate(*calib_ps*, *analyte_ratios=None*)
Apply calibration to data.

The *calib_dict* must be calculated at the *analyse* level, and passed to this calibrate function.

Parameters **calib_dict** (*dict*) – A dict of calibration values to apply to each analyte.

Returns

Return type None

correct_spectral_interference(*target_analyte*, *source_analyte*, *f*)
Correct spectral interference.

Subtract interference counts from target_analyte, based on the intensity of a source_analayte and a known fractional contribution (f).

Correction takes the form: target_analyte -= source_analyte * f

Only operates on background-corrected data ('bkgsub').

To undo a correction, rerun *self.bkg_subtract()*.

Parameters

- **target_analyte** (*str*) – The name of the analyte to modify.
- **source_analyte** (*str*) – The name of the analyte to base the correction on.
- **f** (*float*) – The fraction of the intensity of the source_analyte to subtract from the target_analyte. Correction is: target_analyte - source_analyte * f

Returns

Return type None

correlation_plot(*x_analyte*, *y_analyte*, *window=15*, *filt=True*, *recalc=False*)
Plot the local correlation between two analytes.

Parameters

- **y_analyte** (*x_analyte,*) – The names of the x and y analytes to correlate.
- **window** (*int, None*) – The rolling window used when calculating the correlation.
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.
- **recalc** (*bool*) – If True, the correlation is re-calculated, even if it is already present.

Returns **fig, axs**

Return type figure and axes objects

crossplot(*analytes=None*, *bins=25*, *lognorm=True*, *filt=True*, *colourful=True*, *figsize=(12, 12)*)
Plot analytes against each other.

Parameters

- **analytes** (*optional, array_like or str*) – The analyte(s) to plot. Defaults to all analytes.

- **lognorm** (*bool*) – Whether or not to log normalise the colour scale of the 2D histogram.
- **bins** (*int*) – The number of bins in the 2D histogram.
- **filt** (*str, dict or bool*) – Either logical filter expression contained in a str, a dict of expressions specifying the filter string to use for each analyte or a boolean. Passed to *grab_filt*.

Returns

Return type (fig, axes)

crossplot_filters(*filter_string*, *analytes=None*)
Plot the results of a group of filters in a crossplot.

Parameters

- **filter_string** (*str*) – A string that identifies a group of filters. e.g. 'test' would plot all filters with 'test' in the name.
- **analytes** (*optional, array_like or str*) – The analyte(s) to plot. Defaults to all analytes.

Returns

Return type fig, axes objects

despike(*expdecay_despiker=True*, *exponent=None*, *noise_despiker=True*, *win=3*, *nlim=12.0*, *maxiter=3*)
Applies expdecay_despiker and noise_despiker to data.

Parameters

- **expdecay_despiker** (*bool*) – Whether or not to apply the exponential decay filter.
- **exponent** (*None or float*) – The exponent for the exponential decay filter. If None, it is determined automatically using *find_expocoef*.
- **noise_despiker** (*bool*) – Whether or not to apply the standard deviation spike filter.
- **win** (*int*) – The rolling window over which the spike filter calculates the trace statistics.
- **nlim** (*float*) – The number of standard deviations above the rolling mean that data are excluded.
- **maxiter** (*int*) – The max number of times that the fitler is applied.

Returns

Return type None

filt_nremoved(*filt=True*)

filter_clustering(*analytes*, *filt=False*, *normalise=True*, *method='meanshift'*, *include_time=False*, *sort=None*, *min_data=10*, ***kwargs*)
Applies an n - dimensional clustering filter to the data.

Available Clustering Algorithms

- 'meanshift': The *sklearn.cluster.MeanShift* algorithm. Automatically determines number of clusters in data based on the *bandwidth* of expected variation.
- 'kmeans': The *sklearn.cluster.KMeans* algorithm. Determines the characteristics of a known number of clusters within the data. Must provide *n_clusters* to specify the expected number of clusters.

- 'DBSCAN': The *sklearn.cluster.DBSCAN* algorithm. Automatically determines the number and characteristics of clusters within the data based on the 'connectivity' of the data (i.e. how far apart each data point is in a multi - dimensional parameter space). Requires you to set *eps*, the minimum distance point must be from another point to be considered in the same cluster, and *min_samples*, the minimum number of points that must be within the minimum distance for it to be considered a cluster. It may also be run in automatic mode by specifying *n_clusters* alongside *min_samples*, where eps is decreased until the desired number of clusters is obtained.

For more information on these algorithms, refer to the documentation.

Parameters

- **analytes** (*str*) – The analyte(s) that the filter applies to.
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.
- **normalise** (*bool*) – Whether or not to normalise the data to zero mean and unit variance. Reccomended if clustering based on more than 1 analyte. Uses *sklearn.preprocessing.scale*.
- **method** (*str*) – Which clustering algorithm to use (see above).
- **include_time** (*bool*) – Whether or not to include the Time variable in the clustering analysis. Useful if you're looking for spatially continuous clusters in your data, i.e. this will identify each spot in your analysis as an individual cluster.
- **sort** (*bool, str or array-like*) – Whether or not to label the resulting clusters according to their contents. If used, the cluster with the lowest values will be labelled from 0, in order of increasing cluster mean value.analytes. The sorting rules depend on the value of 'sort', which can be the name of a single analyte (str), a list of several analyte names (array-like) or True (bool), to specify all analytes used to calcualte the cluster.
- **min_data** (*int*) – The minimum number of data points that should be considered by the filter. Default = 10.
- ****kwargs** – Parameters passed to the clustering algorithm specified by *method*.
- **Parameters** (*DBSCAN*) –
- **----------** –
- **bandwidth** (*str or float*) – The bandwith (float) or bandwidth method ('scott' or 'silverman') used to estimate the data bandwidth.
- **bin_seeding** (*bool*) – Modifies the behaviour of the meanshift algorithm. Refer to sklearn.cluster.meanshift documentation.
- **- Means Parameters** (*K*) –
- **----------** –
- **n_clusters** (*int*) – The number of clusters expected in the data.
- **Parameters** –
- **----------** –
- **eps** (*float*) – The minimum 'distance' points must be apart for them to be in the same cluster. Defaults to 0.3. Note: If the data are normalised (they should be for DBSCAN) this is in terms of total sample variance. Normalised data have a mean of 0 and a variance of 1.

- **min_samples** (*int*) – The minimum number of samples within distance *eps* required to be considered as an independent cluster.
- **n_clusters** – The number of clusters expected. If specified, *eps* will be incrementally reduced until the expected number of clusters is found.
- **maxiter** (*int*) – The maximum number of iterations DBSCAN will run.

Returns

Return type None

filter_correlation(*x_analyte*, *y_analyte*, *window=15*, *r_threshold=0.9*, *p_threshold=0.05*, *filt=True*, *recalc=False*)

Calculate correlation filter.

Parameters

- **y_analyte** (*x_analyte,*) – The names of the x and y analytes to correlate.
- **window** (*int, None*) – The rolling window used when calculating the correlation.
- **r_threshold** (*float*) – The correlation index above which to exclude data. Note: the absolute pearson R value is considered, so negative correlations below *-r_threshold* will also be excluded.
- **p_threshold** (*float*) – The significant level below which data are excluded.
- **filt** (*bool*) – Whether or not to apply existing filters to the data before calculating this filter.
- **recalc** (*bool*) – If True, the correlation is re-calculated, even if it is already present.

Returns

Return type None

filter_exclude_downhole(*threshold*, *filt=True*)

Exclude all points down-hole (after) the first excluded data.

Parameters

- **threhold** (*int*) – The minimum number of contiguous excluded data points that must exist before downhole exclusion occurs.
- **file** (*valid filter string or bool*) – Which filter to consider. If True, applies to currently active filters.

filter_gradient_threshold(*analyte*, *win*, *threshold*, *recalc=True*, *win_mode='mid'*, *win_exclude_outside=True*, *absolute_gradient=True*)

Apply gradient threshold filter.

Generates threshold filters for the given analytes above and below the specified threshold.

Two filters are created with prefixes '_above' and '_below'. '_above' keeps all the data above the threshold. '_below' keeps all the data below the threshold.

i.e. to select data below the threshold value, you should turn the '_above' filter off.

Parameters

- **analyte** (*str*) – Description of *analyte*.
- **threshold** (*float*) – Description of *threshold*.
- **win** (*int*) – Window used to calculate gradients (n points)
- **recalc** (*bool*) – Whether or not to re-calculate the gradients.

- **win_mode** (*str*) – Whether the rolling window should be centered on the left, middle or centre of the returned value. Can be 'left', 'mid' or 'right'.
- **win_exclude_outside** (*bool*) – If True, regions at the start and end where the gradient cannot be calculated (depending on win_mode setting) will be excluded by the filter.
- **absolute_gradient** (*bool*) – If True, the filter is applied to the absolute gradient (i.e. always positive), allowing the selection of 'flat' vs 'steep' regions regardless of slope direction. If Falose, the sign of the gradient matters, allowing the selection of positive or negative slopes only.

Returns

Return type None

filter_new(*name*, *filt_str*)

Make new filter from combination of other filters.

Parameters

- **name** (*str*) – The name of the new filter. Should be unique.
- **filt_str** (*str*) – A logical combination of partial strings which will create the new filter. For example, 'Albelow & Mnbelow' will combine all filters that partially match 'Albelow' with those that partially match 'Mnbelow' using the 'AND' logical operator.

Returns

Return type None

filter_report(*filt=None*, *analytes=None*, *savedir=None*, *nbin=5*)

Visualise effect of data filters.

Parameters

- **filt** (*str*) – Exact or partial name of filter to plot. Supports partial matching. i.e. if 'cluster' is specified, all filters with 'cluster' in the name will be plotted. Defaults to all filters.
- **analyte** (*str*) – Name of analyte to plot.
- **save** (*str*) – file path to save the plot

Returns

Return type (fig, axes)

filter_threshold(*analyte*, *threshold*)

Apply threshold filter.

Generates threshold filters for the given analytes above and below the specified threshold.

Two filters are created with prefixes '_above' and '_below'. '_above' keeps all the data above the threshold. '_below' keeps all the data below the threshold.

i.e. to select data below the threshold value, you should turn the '_above' filter off.

Parameters

- **analyte** (*TYPE*) – Description of *analyte*.
- **threshold** (*TYPE*) – Description of *threshold*.

Returns

Return type None

filter_trim(*start=1*, *end=1*, *filt=True*)
Remove points from the start and end of filter regions.

Parameters

- **end** (*start,*) – The number of points to remove from the start and end of the specified filter.
- **filt** (*valid filter string or bool*) – Which filter to trim. If True, applies to currently active filters.

get_params()
Returns paramters used to process data.

Returns dict of analysis parameters

Return type dict

gplot(*analytes=None, win=5, figsize=[10, 4], filt=False, recalc=False, ranges=False, focus_stage=None, ax=None*)
Plot analytes gradients as a function of Time.

Parameters

- **analytes** (*array_like*) – list of strings containing names of analytes to plot. None = all analytes.
- **win** (*int*) – The window over which to calculate the rolling gradient.
- **figsize** (*tuple*) – size of final figure.
- **ranges** (*bool*) – show signal/background regions.

Returns

Return type figure, axis

mkrngs()
Transform boolean arrays into list of limit pairs.

Gets Time limits of signal/background boolean arrays and stores them as sigrng and bkgrng arrays. These arrays can be saved by 'save_ranges' in the analyse object.

optimisation_plot(*overlay_alpha=0.5*, ***kwargs*)
Plot the result of signal_optimise.

signal_optimiser must be run first, and the output stored in the *opt* attribute of the latools.D object.

Parameters

- **d** (*latools.D object*) – A latools data object.
- **overlay_alpha** (*float*) – The opacity of the threshold overlays. Between 0 and 1.
- ****kwargs** – Passed to *tplot*

ratio(*internal_standard=None*, *analytes=None*)
Divide all analytes by a specified internal_standard analyte.

Parameters **internal_standard** (*str*) – The analyte used as the internal_standard.

Returns

Return type None

sample_stats (*analytes=None*, *filt=True*, *stat_fns={}*, *eachtrace=True*, *focus_stage=None*)
Calculate sample statistics

Returns samples, analytes, and arrays of statistics of shape (samples, analytes). Statistics are calculated from the 'focus' data variable, so output depends on how the data have been processed.

Parameters

- **analytes** (*array_like*) – List of analytes to calculate the statistic on
- **filt** (*bool or str*) –

  The filter to apply to the data when calculating sample statistics. bool: True applies filter specified in filt.switches. str: logical string specifying a partucular filter
- **stat_fns** (*dict*) – Dict of {name: function} pairs. Functions that take a single array_like input, and return a single statistic. Function should be able to cope with NaN values.
- **eachtrace** (*bool*) – True: per - ablation statistics False: whole sample statistics

Returns

Return type None

setfocus (*focus*)
Set the 'focus' attribute of the data file.

The 'focus' attribute of the object points towards data from a particular stage of analysis. It is used to identify the 'working stage' of the data. Processing functions operate on the 'focus' stage, so if steps are done out of sequence, things will break.

Names of analysis stages:

- 'rawdata': raw data, loaded from csv file when object is initialised.
- 'despiked': despiked data.
- 'signal'/'background': isolated signal and background data, padded with np.nan. Created by self.separate, after signal and background regions have been identified by self.autorange.
- 'bkgsub': background subtracted data, created by self.bkg_correct
- 'ratios': element ratio data, created by self.ratio.
- 'calibrated': ratio data calibrated to standards, created by self.calibrate.

Parameters **focus** (*str*) – The name of the analysis stage desired.

Returns

Return type None

signal_optimiser (*analytes*, *min_points=5*, *threshold_mode='kde_first_max'*, *threshold_mult=1.0*, *x_bias=0*, *weights=None*, *filt=True*, *mode='minimise'*)
Optimise data selection based on specified analytes.

Identifies the longest possible contiguous data region in the signal where the relative standard deviation (std) and concentration of all analytes is minimised.

Optimisation is performed via a grid search of all possible contiguous data regions. For each region, the mean std and mean scaled analyte concentration ('amplitude') are calculated.

The size and position of the optimal data region are identified using threshold std and amplitude values. Thresholds are derived from all calculated stds and amplitudes using the method specified by *threshold_mode*. For example, using the 'kde_max' method, a probability density function (PDF) is calculated

for std and amplitude values, and the threshold is set as the maximum of the PDF. These thresholds are then used to identify the size and position of the longest contiguous region where the std is below the threshold, and the amplitude is either below the threshold.

All possible regions of the data that have at least *min_points* are considered.

For a graphical demonstration of the action of signal_optimiser, use *optimisation_plot*.

Parameters

- **d** (*latools.D object*) – An latools data object.
- **analytes** (*str or array-like*) – Which analytes to consider.
- **min_points** (*int*) – The minimum number of contiguous points to consider.
- **threshold_mode** (*str*) – The method used to calculate the optimisation thresholds. Can be 'mean', 'median', 'kde_max' or 'bayes_mvs', or a custom function. If a function, must take a 1D array, and return a single, real number.
- **weights** (*array-like of length len(analytes)*) – An array of numbers specifying the importance of each analyte considered. Larger number makes the analyte have a greater effect on the optimisation. Default is None.

tplot (*analytes=None, figsize=[10, 4], scale='log', filt=None, ranges=False, stats=False, stat='nanmean', err='nanstd', focus_stage=None, err_envelope=False, ax=None*)
Plot analytes as a function of Time.

Parameters

- **analytes** (*array_like*) – list of strings containing names of analytes to plot. None = all analytes.
- **figsize** (*tuple*) – size of final figure.
- **scale** (*str or None*) – 'log' = plot data on log scale
- **filt** (*bool, str or dict*) – False: plot unfiltered data. True: plot filtered data over unfiltered data. str: apply filter key to all analytes dict: apply key to each analyte in dict. Must contain all analytes plotted. Can use self.filt.keydict.
- **ranges** (*bool*) – show signal/background regions.
- **stats** (*bool*) – plot average and error of each trace, as specified by *stat* and *err*.
- **stat** (*str*) – average statistic to plot.
- **err** (*str*) – error statistic to plot.

Returns

Return type figure, axis

2.1.3 Filtering

class latools.filtering.filt_obj.**filt** (*size*, *analytes*)
Bases: object

add (*name*, *filt*, *info=''*, *params=()*, *setn=None*)

clean ()

clear ()

fuzzmatch (*fuzzkey*, *multi=True*)

get_components(*analyte*)

get_info()

grab_filt(*filt*, *analyte=None*)

make_analyte(*analyte*)

make_fromkey(*key*)

make_keydict(*analyte=None*)

off(*analyte=None*, *filt=None*)

on(*analyte=None*, *filt=None*)

remove(*name=None*, *setn=None*)

Functions for automatic selection optimisation.

latools.filtering.signal_optimiser.**bayes_scale**(*s*)
Remove mean and divide by standard deviation, using bayes_kvm statistics.

latools.filtering.signal_optimiser.**calc_window_mean_std**(*s*, *min_points*, *ind=None*)
Apply fn to all contiguous regions in s that have at least min_points.

latools.filtering.signal_optimiser.**calc_windows**(*fn*, *s*, *min_points*)
Apply fn to all contiguous regions in s that have at least min_points.

latools.filtering.signal_optimiser.**calculate_optimisation_stats**(*d*, *analytes*, *min_points*, *weights*, *ind*, *x_bias=0*)

latools.filtering.signal_optimiser.**median_scaler**(*s*)
Remove median, divide by IQR.

latools.filtering.signal_optimiser.**optimisation_plot**(*d*, *overlay_alpha=0.5*, ***kwargs*)
Plot the result of signal_optimise.

signal_optimiser must be run first, and the output stored in the *opt* attribute of the latools.D object.

Parameters

- **d** (*latools.D object*) – A latools data object.
- **overlay_alpha** (*float*) – The opacity of the threshold overlays. Between 0 and 1.
- ****kwargs** – Passed to *tplot*

latools.filtering.signal_optimiser.**scale**(*s*)
Remove the mean, and divide by the standard deviation.

latools.filtering.signal_optimiser.**scaler**(*s*)
Remove median, divide by IQR.

latools.filtering.signal_optimiser.**signal_optimiser**(*d*, *analytes*, *min_points=5*, *threshold_mode='kde_first_max'*, *threshold_mult=1.0*, *x_bias=0*, *weights=None*, *ind=None*, *mode='minimise'*)
Optimise data selection based on specified analytes.

Identifies the longest possible contiguous data region in the signal where the relative standard deviation (std) and concentration of all analytes is minimised.

Optimisation is performed via a grid search of all possible contiguous data regions. For each region, the mean std and mean scaled analyte concentration ('amplitude') are calculated.

The size and position of the optimal data region are identified using threshold std and amplitude values. Thresholds are derived from all calculated stds and amplitudes using the method specified by *threshold_mode*. For example, using the 'kde_max' method, a probability density function (PDF) is calculated for std and amplitude values, and the threshold is set as the maximum of the PDF. These thresholds are then used to identify the size and position of the longest contiguous region where the std is below the threshold, and the amplitude is either below the threshold.

All possible regions of the data that have at least *min_points* are considered.

For a graphical demonstration of the action of signal_optimiser, use *optimisation_plot*.

Parameters

- **d** (*latools.D object*) – An latools data object.
- **analytes** (*str or array-like*) – Which analytes to consider.
- **min_points** (*int*) – The minimum number of contiguous points to consider.
- **threshold_mode** (*str*) – The method used to calculate the optimisation thresholds. Can be 'mean', 'median', 'kde_max' or 'bayes_mvs', or a custom function. If a function, must take a 1D array, and return a single, real number.
- **threshood_mult** (*float or tuple*) – A multiplier applied to the calculated threshold before use. If a tuple, the first value is applied to the mean threshold, and the second is applied to the standard deviation threshold. Reduce this to make data selection more stringent.
- **x_bias** (*float*) – If non-zero, a bias is applied to the calculated statistics to prefer the beginning (if > 0) or end (if < 0) of the signal. Should be between zero and 1.
- **weights** (*array-like of length len(analytes)*) – An array of numbers specifying the importance of each analyte considered. Larger number makes the analyte have a greater effect on the optimisation. Default is None.
- **ind** (*boolean array*) – A boolean array the same length as the data. Where false, data will not be included.
- **mode** (*str*) – Whether to 'minimise' or 'maximise' the concentration of the elements.

Returns dict, str

Return type optimisation result, error message

class latools.filtering.classifier_obj.**classifier**(*analytes*, *sort_by=0*)
Bases: object

fit(*data*, *method='kmeans'*, ***kwargs*)
fit classifiers from large dataset.

Parameters

- **data** (*dict*) – A dict of data for clustering. Must contain items with the same name as analytes used for clustering.
- **method** (*str*) – A string defining the clustering method used. Can be:
  - 'kmeans' : K-Means clustering algorithm
  - 'meanshift' : Meanshift algorithm
- **n_clusters** (*int*) – *K-Means only*. The numebr of clusters to identify

- **bandwidth** (*float*) – *Meanshift only.* The bandwidth value used during clustering. If none, determined automatically. Note: the data are scaled before clutering, so this is not in the same units as the data.
- **bin_seeding** (*bool*) – *Meanshift only.* Whether or not to use 'bin_seeding'. See documentation for *sklearn.cluster.MeanShift*.
- ****kwargs** – passed to *sklearn.cluster.MeanShift*.

Returns

Return type list

fit_kmeans(*data*, *n_clusters*, ***kwargs*)
Fit KMeans clustering algorithm to data.

Parameters

- **data** (*array-like*) – A dataset formatted by *classifier.fitting_data*.
- **n_clusters** (*int*) – The number of clusters in the data.
- ****kwargs** – passed to *sklearn.cluster.KMeans*.

Returns

Return type Fitted *sklearn.cluster.KMeans* object.

fit_meanshift(*data*, *bandwidth=None*, *bin_seeding=False*, ***kwargs*)
Fit MeanShift clustering algorithm to data.

Parameters

- **data** (*array-like*) – A dataset formatted by *classifier.fitting_data*.
- **bandwidth** (*float*) – The bandwidth value used during clustering. If none, determined automatically. Note: the data are scaled before clutering, so this is not in the same units as the data.
- **bin_seeding** (*bool*) – Whether or not to use 'bin_seeding'. See documentation for *sklearn.cluster.MeanShift*.
- ****kwargs** – passed to *sklearn.cluster.MeanShift*.

Returns

Return type Fitted *sklearn.cluster.MeanShift* object.

fitting_data(*data*)
Function to format data for cluster fitting.

Parameters **data** (*dict*) – A dict of data, containing all elements of *analytes* as items.

Returns

Return type A data array for initial cluster fitting.

format_data(*data*, *scale=True*)
Function for converting a dict to an array suitable for sklearn.

Parameters

- **data** (*dict*) – A dict of data, containing all elements of *analytes* as items.
- **scale** (*bool*) – Whether or not to scale the data. Should always be *True*, unless used by *classifier.fitting_data* where a scaler hasn't been created yet.

Returns

Return type A data array suitable for use with *sklearn.cluster*.

map_clusters(*size*, *sampled*, *clusters*)
Translate cluster identity back to original data size.

Parameters

- **size** (*int*) – size of original dataset
- **sampled** (*array-like*) – integer array describing location of finite values in original data.
- **clusters** (*array-like*) – integer array of cluster identities

Returns

- *list of cluster identities the same length as original*
- *data. Where original data are non-finite, returns -2.*

predict(*data*)
Label new data with cluster identities.

Parameters

- **data** (*dict*) – A data dict containing the same analytes used to fit the classifier.
- **sort_by** (*str*) – The name of an analyte used to sort the resulting clusters. If None, defaults to the first analyte used in fitting.

Returns

Return type array of clusters the same length as the data.

sort_clusters(*data*, *cs*, *sort_by*)
Sort clusters by the concentration of a particular analyte.

Parameters

- **data** (*dict*) – A dataset containing sort_by as a key.
- **cs** (*array-like*) – An array of clusters, the same length as values of data.
- **sort_by** (*str*) – analyte to sort the clusters by

Returns

Return type array of clusters, sorted by mean value of sort_by analyte.

2.1.4 Configuration

Note: the entire config module is available at the top level (i.e. `latools.config`).

Functions to help configure LAtools.

(c) Oscar Branson : https://github.com/oscarbranson

latools.helpers.config.**change_default**(*config*)
Change the default configuration.

latools.helpers.config.**config_locator**()
Returns the path to the file containing your LAtools configurations.

latools.helpers.config.**copy_SRM_file**(*destination=None*, *config='DEFAULT'*)
Creates a copy of the default SRM table at the specified location.

Parameters

- **destination** (*str*) – The save location for the SRM file. If no location specified, saves it as 'LAtools_[config]_SRMTable.csv' in the current working directory.
- **config** (*str*) – It's possible to set up different configurations with different SRM files. This specifies the name of the configuration that you want to copy the SRM file from. If not specified, the 'DEFAULT' configuration is used.

latools.helpers.config.**create**(*config_name*, *srmfile=None*, *dataformat=None*, *base_on='DEFAULT'*, *make_default=False*)

Adds a new configuration to latools.cfg.

Parameters

- **config_name** (*str*) – The name of the new configuration. This should be descriptive (e.g. UC Davis Foram Group)
- **srmfile** (*str (optional)*) – The location of the srm file used for calibration.
- **dataformat** (*str (optional)*) – The location of the dataformat definition to use.
- **base_on** (*str*) – The name of the existing configuration to base the new one on. If either srm_file or dataformat are not specified, the new config will copy this information from the base_on config.
- **make_default** (*bool*) – Whether or not to make the new configuration the default for future analyses. Default = False.

Returns

Return type None

latools.helpers.config.**delete**(*config*)

latools.helpers.config.**get_dataformat_template**(*destination='./LAtools_dataformat_template.json'*)

Copies a data format description JSON template to the specified location.

latools.helpers.config.**locate**()

Prints and returns the location of the latools.cfg file.

latools.helpers.config.**print_all**()

Prints all currently defined configurations.

latools.helpers.config.**read_configuration**(*config='DEFAULT'*)

Read LAtools configuration file, and return parameters as dict.

latools.helpers.config.**read_latoolscfg**()

Reads configuration, returns a ConfigParser object.

Distinct from read_configuration, which returns a dict.

latools.helpers.config.**test_dataformat**(*data_file*, *dataformat_file*, *name_mode='file_names'*)

Test a data formatfile against a particular data file.

This goes through all the steps of data import printing out the results of each step, so you can see where the import fails.

Parameters

- **data_file** (*str*) – Path to data file, including extension.
- **dataformat** (*dict or str*) – A dataformat dict, or path to file. See example below.
- **name_mode** (*str*) – How to identyfy sample names. If 'file_names' uses the input name of the file, stripped of the extension. If 'metadata_names' uses the 'name' attribute of the 'meta' sub-dictionary in dataformat. If any other str, uses this str as the sample name.

Example

```
>>>
{'genfromtext_args': {'delimiter': ',',
                      'skip_header': 4},  # passed directly to np.genfromtxt
 'column_id': {'name_row': 3,  # which row contains the column names
               'delimiter': ',',  # delimeter between column names
               'timecolumn': 0,  # which column contains the 'time' variable
               'pattern': '([A-z]{1,2}[0-9]{1,3})'},  # a regex pattern which␣
↪captures the column names
 'meta_regex': {  # a dict of (line_no: ([descriptors], [regexs])) pairs
                0: (['path'], '(.*)'),
                2: (['date', 'method'],  # MUST include date
                 '([A-Z][a-z]+ [0-9]+ [0-9]{4}[ ]+[0-9:]+ [amp]+).* ([A-z0-9]+\.m)
↪')
               }
}
```

Returns sample, analytes, data, meta

Return type tuple

`latools.helpers.config.`**`update`**(*config*, *parameter*, *new_value*)

2.1.5 Preprocessing

Functions for splitting long files into multiple short ones.

(c) Oscar Branson : https://github.com/oscarbranson

`latools.preprocessing.split.`**`by_regex`**(*file*, *outdir=None*, *split_pattern=None*, *global_header_rows=0*, *fname_pattern=None*, *trim_tail_lines=0*, *trim_head_lines=0*)

Split one long analysis file into multiple smaller ones.

Parameters

- **`file`** (*`str`*) – The path to the file you want to split.
- **`outdir`** (*`str`*) – The directory to save the split files to. If None, files are saved to a new directory called 'split', which is created inside the data directory.
- **`split_pattern`** (*`regex string`*) – A regular expression that will match lines in the file that mark the start of a new section. Does not have to match the whole line, but must provide a positive match to the lines containing the pattern.
- **`global_header_rows`** (*`int`*) – How many rows at the start of the file to include in each new sub-file.
- **`fname_pattern`** (*`regex string`*) – A regular expression that identifies a new file name in the lines identified by split_pattern. If none, files will be called 'noname_N'. The extension of the main file will be used for all sub-files.
- **`trim_head_lines`** (*`int`*) – If greater than zero, this many lines are removed from the start of each segment
- **`trim_tail_lines`** (*`int`*) – If greater than zero, this many lines are removed from the end of each segment

Returns Path to new directory

Return type str

latools.preprocessing.split.**long_file**(*data_file*, *dataformat*, *sample_list*, *analyte='total_counts'*, *savedir=None*, *srm_id=None*, *combine_same_name=True*, *defrag_to_match_sample_list=True*, *min_points=0*, *plot=True*, ***autorange_args*)

Split single long files containing multiple analyses into multiple files containing single analyses.

Imports a long datafile and uses *latools.processes.autorange* to identify ablations in the long file based on your chosen analyte. The data are then saved as multiple files each containing a single ablation, named using the list of names you provide.

Data will be saved in latools' 'REPRODUCE' format.

WARNING: This functionality is currently *very beta*. Use carefully.

TODO: Check for existing files in savedir, don't overwrite?

Parameters

- **data_file** (*str*) – The path to the data file you want to read.
- **dataformat** (*dataformat dict*) – A valid dataformat dict. See online documentation for more details.
- **sample_list** (*array-like*) – A list of strings that will be used to name the individual files. One sample name can contain a 'wildcard' character '+' or '*'. If we find more ablations than the number of names in sample_list, we'll expand this wildcard to name each unlabelled ablation. If the wildcard is '+' the abltations are given unique numbered names and split up into separate files, whereas for '*' the ablations are given the same and saved into a single file. One *one* sample name can contain a wildcard.
- **analyte** (*str*) – The analyte that autorange uses to identify ablations. Can be any valid analyte in the data. Defaults to 'total_counts'.
- **savedir** (*str*) – The directory to save the data in. Defaults to the name of the data_file, appended with '_split'.
- **srm_id** (*str*) – If given, all file names containing srm_id will be replaced with srm_id.
- ****autorange_args** – Additional arguments passed to la.processes.autorange used for identifying ablations.

Returns

Return type None

latools.preprocessing.split.**plot_long_file_split**(*dat*, *sig*, *bkg*, *sections*)

2.1.6 Helpers

Function for reading LA-ICPMS data files.

(c) Oscar Branson : https://github.com/oscarbranson

latools.processes.data_read.**read_data**(*data_file*, *dataformat*, *name_mode*)

Load data_file described by a dataformat dict.

Parameters

- **data_file** (*str*) – Path to data file, including extension.
- **dataformat** (*dict*) – A dataformat dict, see example below.

- **name_mode** (*str*) – How to identyfy sample names. If 'file_names' uses the input name of the file, stripped of the extension. If 'metadata_names' uses the 'name' attribute of the 'meta' sub-dictionary in dataformat. If any other str, uses this str as the sample name.

Example

```
>>>
{'genfromtext_args': {'delimiter': ',',
                      'skip_header': 4},  # passed directly to np.genfromtxt
 'column_id': {'name_row': 3,  # which row contains the column names
               'delimiter': ',',  # delimeter between column names
               'timecolumn': 0,  # which column contains the 'time' variable
               'pattern': '([A-z]{1,2}[0-9]{1,3})'},  # a regex pattern which␣
↪captures the column names
 'meta_regex': {  # a dict of (line_no: ([descriptors], [regexs])) pairs
                "0": (['path'], '(.*)'),
                "2": (['date', 'method'],  # should include date
                 '([A-Z][a-z]+ [0-9]+ [0-9]{4}[ ]+[0-9:]+ [amp]+).* ([A-z0-9]+\.m)
↪')
               }
}
```

Returns sample, analytes, data, meta

Return type tuple

latools.processes.data_read.**read_dataformat**(*file*)

Reads a dataformat .json file and returns it as a dict.

Parameters **file** (*str*) – Path to dataformat.json file.

Functions for de-spiking LA-ICPMS data (outlier removal).

(c) Oscar Branson : https://github.com/oscarbranson

latools.processes.despiking.**expdecay_despike**(*sig*, *expdecay_coef*, *tstep*, *maxiter=3*)

Apply exponential decay filter to remove physically impossible data based on instrumental washout.

The filter is re-applied until no more points are removed, or maxiter is reached.

Parameters

- **exponent** (*float*) – Exponent used in filter
- **tstep** (*float*) – The time increment between data points.
- **maxiter** (*int*) – The maximum number of times the filter should be applied.

Returns

Return type None

latools.processes.despiking.**noise_despike**(*sig*, *win=3*, *nlim=24.0*, *maxiter=4*)

Apply standard deviation filter to remove anomalous values.

Parameters

- **win** (*int*) – The window used to calculate rolling statistics.
- **nlim** (*float*) – The number of standard deviations above the rolling mean above which data are considered outliers.

Returns

Return type None

Functions for automatically distinguishing between signal and background in LA-ICPMS data.

(c) Oscar Branson : https://github.com/oscarbranson

latools.processes.signal_id.**autorange**(*xvar*, *sig*, *gwin=7*, *swin=None*, *win=30*, *on_mult=(1.5, 1.0)*, *off_mult=(1.0, 1.5)*, *nbin=10*, *transform='log'*, *thresh=None*)

Automatically separates signal and background in an on/off data stream.

Step 1: Thresholding. KMeans clustering is used to identify data where the laser is 'on' vs where the laser is 'off'.

Step 2: Transition Removal. The width of the transition regions between signal and background are then determined, and the transitions are excluded from analysis. The width of the transitions is determined by fitting a gaussian to the smoothed first derivative of the analyte trace, and determining its width at a point where the gaussian intensity is at at *conf* time the gaussian maximum. These gaussians are fit to subsets of the data centered around the transitions regions determined in Step 1, +/- *win* data points. The peak is further isolated by finding the minima and maxima of a second derivative within this window, and the gaussian is fit to the isolated peak.

Parameters

- **xvar** (*array-like*) – Independent variable (usually time).
- **sig** (*array-like*) – Dependent signal, of shape (nsamples, nfeatures). Should be clear distinction between laser 'on' and 'off' regions.
- **gwin** (*int*) – The window used for calculating first derivative. Defaults to 7.
- **swin** (*int*) – The window used for signal smoothing. If None, `gwin // 2`.
- **win** (*int*) – The width (c +/- win) of the transition data subsets. Defaults to 20.
- **and off_mult** (*on_mult*) – Control the width of the excluded transition regions, which is defined relative to the peak full-width-half-maximum (FWHM) of the transition gradient. The region n * FHWM below the transition, and m * FWHM above the tranision will be excluded, where (n, m) are specified in *on_mult* and *off_mult*. *on_mult* and *off_mult* apply to the off-on and on-off transitions, respectively. Defaults to (1.5, 1) and (1, 1.5).
- **transform** (*str*) – How to transform the data. Default is 'log'.

Returns **fbkg, fsig, ftrn, failed** – where fbkg, fsig and ftrn are boolean arrays the same length as sig, that are True where sig is background, signal and transition, respecively. failed contains a list of transition positions where gaussian fitting has failed.

Return type tuple

latools.processes.signal_id.**autorange_components**(*t*, *sig*, *transform='log'*, *gwin=7*, *swin=None*, *win=30*, *on_mult=(1.5, 1.0)*, *off_mult=(1.0, 1.5)*, *thresh=None*)

Returns the components underlying the autorange algorithm.

Returns

- **t** (*array-like*) – Time axis (independent variable)
- **sig** (*array-like*) – Raw signal (dependent variable)
- **sigs** (*array-like*) – Smoothed signal (swin)

- **tsig** (*array-like*) – Transformed raw signal (transform)
- **tsigs** (*array-like*) – Transformed smoothed signal (transform, swin)
- **kde_x** (*array-like*) – kernel density estimate of smoothed signal.
- **yd** (*array-like*) – bins of kernel density estimator.
- **g** (*array-like*) – gradient of smoothed signal (swin, gwin)
- **trans** (*dict*) – per-transition data.
- **thresh** (*float*) – threshold identified from kernel density plot

latools.processes.signal_id.**log_nozero**(*a*, ***kwargs*)

latools.processes.signal_id.**separate_signal**(*X*, *transform=None*, *scaleX=True*)

Helper functions used by multiple parts of LAtools.

(c) Oscar Branson : https://github.com/oscarbranson

class latools.helpers.helpers.**Bunch**(**args*, ***kwds*)

Bases: dict

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*], ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

latools.helpers.helpers.**bool_2_indices**(*a*)
Convert boolean array into a 2D array of (start, stop) pairs.

latools.helpers.helpers.**bool_transitions**(*a*)
Return indices where a boolean array changes from True to False

latools.helpers.helpers.**calc_grads**(*x*, *dat*, *keys=None*, *win=5*, *win_mode='mid'*)
Calculate gradients of values in dat.

Parameters

- **x** (*array like*) – Independent variable for items in dat.

- **dat** (*dict*) – {key: dependent_variable} pairs
- **keys** (*str or array-like*) – Which keys in dict to calculate the gradient of.
- **win** (*int*) – The side of the rolling window for gradient calculation
- **win_mode** (*str*) – Describes the jusitification of the rolling window relative to the returned values. Can be 'left', 'mid' or 'right'.

Returns

Return type dict of gradients

latools.helpers.helpers.**collate_data**(*in_dir*, *extension='.csv'*, *out_dir=None*)

Copy all csvs in nested directroy to single directory.

Function to copy all csvs from a directory, and place them in a new directory.

Parameters

- **in_dir** (*str*) – Input directory containing csv files in subfolders
- **extension** (*str*) – The extension that identifies your data files. Defaults to '.csv'.
- **out_dir** (*str*) – Destination directory

Returns

Return type None

latools.helpers.helpers.**enumerate_bool**(*bool_array*, *nstart=0*)

Consecutively numbers contiguous booleans in array.

i.e. a boolean sequence, and resulting numbering T F T T T F T F F F T T F 0-1 1 1 - 2 —3 3 -

where ' - '

Parameters

- **bool_array** (*array_like*) – Array of booleans.
- **nstart** (*int*) – The number of the first boolean group.

latools.helpers.helpers.**fastgrad**(*a*, *win=11*, *win_mode='mid'*)

Returns rolling - window gradient of a.

Function to efficiently calculate the rolling gradient of a numpy array using 'stride_tricks' to split up a 1D array into an ndarray of sub - sections of the original array, of dimensions [len(a) - win, win].

Parameters

- **a** (*array_like*) – The 1D array to calculate the rolling gradient of.
- **win** (*int*) – The width of the rolling window.
- **win_mode** (*str*) – Describes the jusitification of the rolling window relative to the returned values. Can be 'left', 'mid' or 'right'.

Returns Gradient of a, assuming as constant integer x - scale.

Return type array_like

latools.helpers.helpers.**fastsmooth**(*a*, *win=11*)

Returns rolling - window smooth of a.

Function to efficiently calculate the rolling mean of a numpy array using 'stride_tricks' to split up a 1D array into an ndarray of sub - sections of the original array, of dimensions [len(a) - win, win].

Parameters

- **a** (*array_like*) – The 1D array to calculate the rolling gradient of.
- **win** (*int*) – The width of the rolling window.

Returns Gradient of a, assuming as constant integer x - scale.

Return type array_like

latools.helpers.helpers.**findmins**(*x*, *y*)
Function to find local minima.

Parameters **y** (*x,*) – 1D arrays of the independent (x) and dependent (y) variables.

Returns Array of points in x where y has a local minimum.

Return type array_like

latools.helpers.helpers.**get_date**(*datetime*, *time_format=None*)
Return a datetime oject from a string, with optional time format.

Parameters

- **datetime** (*str*) – Date-time as string in any sensible format.
- **time_format** (*datetime str (optional)*) – String describing the datetime format. If missing uses dateutil.parser to guess time format.

latools.helpers.helpers.**get_example_data**(*destination_dir*)

latools.helpers.helpers.**get_total_n_points**(*d*)
Returns the total number of data points in values of dict.

d : dict

latools.helpers.helpers.**get_total_time_span**(*d*)
Returns total length of analysis.

latools.helpers.helpers.**rangecalc**(*xs*, *pad=0.05*)

latools.helpers.helpers.**rolling_window**(*a*, *window*, *window_mode='mid'*, *pad=None*)
Returns (win, len(a)) rolling - window array of data.

Parameters

- **a** (*array_like*) – Array to calculate the rolling window of
- **window** (*int*) – Description of *window*.
- **window_mode** (*str*) – Describes the jusitification of the rolling window relative to the returned values. Can be 'left', 'mid' or 'right'.
- **pad** (*same as dtype(a)*) – How to pad the ends of the array such that shape[0] of the returned array is the same as len(a). Can be 'ends', 'mean_ends' or 'repeat_ends'. 'ends' just extends the start or end value across all the extra windows. 'mean_ends' extends the mean value of the end windows. 'repeat_ends' repeats the end window to completion.

Returns An array of shape (n, window), where n is either len(a) - window if pad is None, or len(a) if pad is not None.

Return type array_like

latools.helpers.helpers.**stack_keys**(*ddict*, *keys*, *extra=None*)
Combine elements of ddict into an array of shape (len(ddict[key]), len(keys)).

Useful for preparing data for sklearn.

Parameters

- **ddict** (*dict*) – A dict containing arrays or lists to be stacked. Must be of equal length.
- **keys** (*list or str*) – The keys of dict to stack. Must be present in ddict.
- **extra** (*list (optional)*) – A list of additional arrays to stack. Elements of extra must be the same length as arrays in ddict. Extras are inserted as the first columns of output.

latools.helpers.helpers.**tuples_2_bool**(*tuples*, *x*)
Generate boolean array from list of limit tuples.

Parameters

- **tuples** (*array_like*) – [2, n] array of (start, end) values
- **x** (*array_like*) – x scale the tuples are mapped to

Returns boolean array, True where x is between each pair of tuples.

Return type array_like

class latools.helpers.helpers.**un_interp1d**(*x*, *y*, *fill_value=nan*, ***kwargs*)
Bases: object

object for handling interpolation of values with uncertainties.

new(*xn*)

new_nom(*xn*)

new_std(*xn*)

latools.helpers.helpers.**unitpicker**(*a*, *llim=0.1*, *denominator=None*, *focus_stage=None*)
Determines the most appropriate plotting unit for data.

Parameters

- **a** (*float or array-like*) – number to optimise. If array like, the 25% quantile is optimised.
- **llim** (*float*) – minimum allowable value in scaled data.

Returns (multiplier, unit)

Return type (float, str)

Functions for dealing with chemical formulae, and converting between molar ratios and mass fractions.

(c) Oscar Branson : https://github.com/oscarbranson

latools.helpers.chemistry.**analyte_mass**(*analyte*, *in_name=True*)
Returns the mass of a given analyte.

If the name contains a number (e.g. Ca43), that number is returned. If the name contains no number but an element name (e.g. Ca), the average mass of that element is returned.

Parameters

- **analyte** (*str or array-like*) – The name or names of the analytes to be considered.
- **in_name** (*bool*) – If True, numbers in the analyte name are preferred.

latools.helpers.chemistry.**calc_M**(*molecule*)
Returns molecular weight of molecule.

Where molecule is in standard chemical notation, e.g. 'CO2', 'HCO3' or B(OH)4

Returns molecular_weight

Return type float

latools.helpers.chemistry.**decompose_molecule**(*molecule*, *n=1*)
Returns the chemical constituents of the molecule, and their number.

Parameters **molecule** (*str*) – A molecule in standard chemical notation, e.g. 'CO2', 'HCO3' or 'B(OH)4'.

Returns **All elements in molecule with their associated counts**

Return type dict

latools.helpers.chemistry.**elements**(*all_isotopes=True*)
Loads a DataFrame of all elements and isotopes.

Scraped from https://www.webelements.com/

Returns

Return type pandas DataFrame with columns (element, atomic_number, isotope, atomic_weight, percent)

latools.helpers.chemistry.**to_mass_fraction**(*molar_ratio*, *massfrac_denominator*, *numerator_mass*, *denominator_mass*)
Converts per-mass concentrations to molar elemental ratios.

Be careful with units.

Parameters

- **molar_ratio** (*float or array-like*) – The molar ratio of elements.
- **massfrac_denominator** (*float or array-like*) – The mass fraction of the denominator element
- **denominator_mass** (*numerator_mass,*) – The atomic mass of the numerator and denominator.

Returns **float or array-like**

Return type The mass fraction of the numerator element.

latools.helpers.chemistry.**to_molar_ratio**(*massfrac_numerator*, *massfrac_denominator*, *numerator_mass*, *denominator_mass*)
Converts per-mass concentrations to molar elemental ratios.

Be careful with units.

Parameters

- **denominator_mass** (*numerator_mass,*) – The atomic mass of the numerator and denominator.
- **massfrac_denominator** (*massfrac_numerator,*) – The per-mass fraction of the numnerator and denominator.

Returns **float or array-like**

Return type The molar ratio of elements in the material

Functions for calculating statistics and handling uncertainties.

(c) Oscar Branson : https://github.com/oscarbranson

latools.helpers.stat_fns.**H15_mean**(*x*)
Calculate the Huber (H15) Robust mean of x.

For details, see: http://www.cscjp.co.jp/fera/document/ANALYSTVol114Decpgs1693-97_1989.pdf
http://www.rsc.org/images/robust-statistics-technical-brief-6_tcm18-214850.pdf

latools.helpers.stat_fns.**H15_se**(*x*)
Calculate the Huber (H15) Robust standard deviation of x.

For details, see: http://www.cscjp.co.jp/fera/document/ANALYSTVol114Decpgs1693-97_1989.pdf
http://www.rsc.org/images/robust-statistics-technical-brief-6_tcm18-214850.pdf

latools.helpers.stat_fns.**H15_std**(*x*)
Calculate the Huber (H15) Robust standard deviation of x.

For details, see: http://www.cscjp.co.jp/fera/document/ANALYSTVol114Decpgs1693-97_1989.pdf
http://www.rsc.org/images/robust-statistics-technical-brief-6_tcm18-214850.pdf

latools.helpers.stat_fns.**R2calc**(*meas*, *model*, *force_zero=False*)

latools.helpers.stat_fns.**gauss**(*x*, **p*)
Gaussian function.

Parameters

- **x** (*array_like*) – Independent variable.
- ***p** (*parameters unpacked to A, mu, sigma*) – A = amplitude, mu = centre, sigma = width

Returns gaussian descriped by *p.

Return type array_like

latools.helpers.stat_fns.**gauss_weighted_stats**(*x*, *yarray*, *x_new*, *fwhm*)
Calculate gaussian weigted moving mean, SD and SE.

Parameters

- **x** (*array-like*) – The independent variable
- **yarray** (*(n,m) array*) – Where n = x.size, and m is the number of dependent variables to smooth.
- **x_new** (*array-like*) – The new x-scale to interpolate the data
- **fwhm** (*int*) – FWHM of the gaussian kernel.

Returns **(mean, std, se)**

Return type tuple

latools.helpers.stat_fns.**nan_pearsonr**(*x*, *y*)

latools.helpers.stat_fns.**nominal_values**(*a*)

latools.helpers.stat_fns.**std_devs**(*a*)

latools.helpers.stat_fns.**stderr**(*a*)
Calculate the standard error of a.

latools.helpers.stat_fns.**unpack_uncertainties**(*uarray*)
Convenience function to unpack nominal values and uncertainties from an `uncertainties.uarray`.

Returns (nominal_values, std_devs)

Plotting functions.

(c) Oscar Branson : https://github.com/oscarbranson

latools.helpers.plot.**autorange_plot**(*t*, *sig*, *gwin=7*, *swin=None*, *win=30*, *on_mult=(1.5*, *1.0)*, *off_mult=(1.0*, *1.5)*, *nbin=10*, *thresh=None*)

Function for visualising the autorange mechanism.

Parameters

- **t** (*array-like*) – Independent variable (usually time).
- **sig** (*array-like*) – Dependent signal, with distinctive 'on' and 'off' regions.
- **gwin** (*int*) – The window used for calculating first derivative. Defaults to 7.
- **swin** (*int*) – The window ised for signal smoothing. If None, gwin // 2.
- **win** (*int*) – The width (c +/- win) of the transition data subsets. Defaults to 20.
- **and off_mult** (*on_mult*) – Control the width of the excluded transition regions, which is defined relative to the peak full-width-half-maximum (FWHM) of the transition gradient. The region n * FHWM below the transition, and m * FWHM above the tranision will be excluded, where (n, m) are specified in *on_mult* and *off_mult*. *on_mult* and *off_mult* apply to the off-on and on-off transitions, respectively. Defaults to (1.5, 1) and (1, 1.5).
- **nbin** (*ind*) – Used to calculate the number of bins in the data histogram. bins = len(sig) // nbin

Returns

Return type fig, axes

latools.helpers.plot.**calc_nrow**(*n*, *ncol*)

latools.helpers.plot.**calibration_plot**(*self*, *analyte_ratios=None*, *datarange=True*, *loglog=False*, *ncol=3*, *srm_group=None*, *percentile_data_cutoff=85*, *save=True*)

Plot the calibration lines between measured and known SRM values.

Parameters

- **analyte_ratios** (*optional, array_like or str*) – The analyte ratio(s) to plot. Defaults to all analyte ratios.
- **datarange** (*boolean*) – Whether or not to show the distribution of the measured data alongside the calibration curve.
- **loglog** (*boolean*) – Whether or not to plot the data on a log - log scale. This is useful if you have two low standards very close together, and want to check whether your data are between them, or below them.
- **ncol** (*int*) – The number of columns in the plot
- **srm_group** (*int*) – Which groups of SRMs to plot in the analysis.
- **percentile_data_cutoff** (*float*) – The upper percentile of data to display in the histogram.

Returns

Return type (fig, axes)

latools.helpers.plot.**correlation_plot**(*self*, *corr=None*)

latools.helpers.plot.**crossplot**(*dat*, *keys=None*, *lognorm=True*, *bins=25*, *figsize=(12*, *12)*, *colourful=True*, *focus_stage=None*, *denominator=None*, *mode='hist2d'*, *cmap=None*, ***kwargs*)

Plot analytes against each other.

The number of plots is n**2 - n, where n = len(keys).

Parameters

- **dat** (*dict*) – A dictionary of key: data pairs, where data is the same length in each entry.
- **keys** (*optional, array_like or str*) – The keys of dat to plot. Defaults to all keys.
- **lognorm** (*bool*) – Whether or not to log normalise the colour scale of the 2D histogram.
- **bins** (*int*) – The number of bins in the 2D histogram.
- **figsize** (*tuple*) –
- **colourful** (*bool*) –

Returns

Return type (fig, axes)

latools.helpers.plot.**filter_report**(*Data*, *filt=None*, *analytes=None*, *savedir=None*, *nbin=5*)
Visualise effect of data filters.

Parameters

- **filt** (*str*) – Exact or partial name of filter to plot. Supports partial matching. i.e. if 'cluster' is specified, all filters with 'cluster' in the name will be plotted. Defaults to all filters.
- **analyte** (*str*) – Name of analyte to plot.
- **save** (*str*) – file path to save the plot

Returns

Return type (fig, axes)

latools.helpers.plot.**gplot**(*self, analytes=None, win=25, figsize=[10, 4], filt=False, ranges=False, focus_stage=None, ax=None, recalc=True*)
Plot analytes gradients as a function of Time.

Parameters

- **analytes** (*array_like*) – list of strings containing names of analytes to plot. None = all analytes.
- **win** (*int*) – The window over which to calculate the rolling gradient.
- **figsize** (*tuple*) – size of final figure.
- **ranges** (*bool*) – show signal/background regions.

Returns

Return type figure, axis

latools.helpers.plot.**histograms**(*dat*, *keys=None*, *bins=25*, *logy=False*, *cmap=None*, *ncol=4*)
Plot histograms of all items in dat.

Parameters

- **dat** (*dict*) – Data in {key: array} pairs.
- **keys** (*arra-like*) – The keys in dat that you want to plot. If None, all are plotted.
- **bins** (*int*) – The number of bins in each histogram (default = 25)
- **logy** (*bool*) – If true, y axis is a log scale.

- **cmap** (*dict*) – The colours that the different items should be. If None, all are grey.

Returns

Return type fig, axes

latools.helpers.plot.**tplot**(*self, analytes=None, figsize=[10, 4], scale='log', filt=None, ranges=False, stats=False, stat='nanmean', err='nanstd', focus_stage=None, err_envelope=False, ax=None*)

Plot analytes as a function of Time.

Parameters

- **analytes** (*array_like*) – list of strings containing names of analytes to plot. None = all analytes.
- **figsize** (*tuple*) – size of final figure.
- **scale** (*str or None*) – 'log' = plot data on log scale
- **filt** (*bool, str or dict*) – False: plot unfiltered data. True: plot filtered data over unfiltered data. str: apply filter key to all analytes dict: apply key to each analyte in dict. Must contain all analytes plotted. Can use self.filt.keydict.
- **ranges** (*bool*) – show signal/background regions.
- **stats** (*bool*) – plot average and error of each trace, as specified by *stat* and *err*.
- **stat** (*str*) – average statistic to plot.
- **err** (*str*) – error statistic to plot.

Returns

Return type figure, axis

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

l

Index

D

E

F

G

H

I

K

L

M

N

V

Z