gigaanalysis

Release 0.4.2

Alex Hickey

Mar 02, 2023

CONTENTS:

A toolbox for processing data that can be expressed as a dependent and independent variable.

This library provides a collection of classes and functions for analysing datasets which are of the from of one independent and one dependent variable. This is very common in condensed mater physics experiment and gigaanalysis was produced for use in high magnetic field facilities.

CHAPTER

ONE

INTRODUCTION

This library provides a collection of classes and functions for analysing datasets which are of the from of one independent and one dependent variable. This is very common in condensed mater physics experiment and gigaanalysis was produced for use in high magnetic field facilities.

1.1 Layout

It broken into a collection of modules for different uses

- *gigaanalysis.data* This contains the Data class which gigaanalysis is built around. It also contains a few functions for very common uses.
- *gigaanalysis.mfunc* - This contains mathematical functions that are useful to act on data objects. This is broken into four sections applying numpy ufuncs, making Data objects, performing FFTs, and transforming Data objects.
- *gigaanalysis.dset* - For saving, loading, and manipulating collections of Data objects which are referred to datasets.
- *gigaanalysis.fit* - For fitting forms to the data contained in Data objects.
- *gigaanalysis.parse* - Contains functions for collecting all the data contained in datasets together, or distribution of data into Data objects.
- *gigaanalysis.qo* - Functions and classes for analysing quantum osculations experiments.
- *gigaanalysis.contour* - Class for producing contour maps from a data set, using Gaussian processes.
- *gigaanalysis.htsc* - Functions which are useful for studying superconductivity.
- *gigaanalysis.magnetism* - Functions for studying magnetism.
- *gigaanalysis.heatc* - Functions for studying heat capacity of materials.
- *gigaanalysis.diglock* - An implementation of a digital lock in.
- *gigaanalysis.highfield* - A class for processing the data from pulsed magnetic field facilities.
- *gigaanalysis.const* - A few useful scientific constants in different systems of units.

1.2 Installation

In an environment with `python >= 3.7` the following should install gigaanalysis.

```
pip install gigaanalysis
```

1.3 Requirements

This was developed mostly using

- python 3.7.7
- numpy 1.21.2
- pandas 1.3.4
- matplotlib 3.4.3
- h5py 2.10.0

I haven't found any problems with using newer versions of these same dependencies.

CHAPTER

TWO

API

Here is the complete list of modules, with there associated functions and classes found in GigaAnalysis.

2.1 GigaAnalysis - `gigaanalysis`

This is made for processing sweep data from physics experiments.

2.2 GigaAnalysis - Data Type - `gigaanalysis.data`

This one module is imported directly into the *gigaanalysis* namespace, so that the classes and functions here can be accessed directly.

This holds the *Data* class and the functions that will manipulate them. This forms the backbone of the rest of the GigaAnalysis. The point of the *Data* object is to hold sweeps. These are data sets with one independent and one dependant variable, which are super common in experimental physics research. By assuming the data is of this type more assumptions and error checking can be facilitated, and this is what GigaAnalysis aims to take advantage of.

class gigaanalysis.data.**Data**(*values*, *split_y=None*, *strip_sort=False*, *interp_full=None*)

Bases: object

The Data Class

Data object holds the data in the measurements. It works as a simple wrapper of a two column numpy array (numpy.ndarray). The data stored in the object is meant to be interpreted as x is a independent variable and y is dependant variable.

Parameters

- **values** (*numpy.ndarray*) – A two column numpy array with the x data in the first column and the y data in the second. If a second no array is given then the first corresponds to the x data.
- **split_y** (*numpy.ndarray, optional*) – A 1D numpy array containing the y data. If None all the data should be contained in first array.
- **strip_sort** (*bool or {'strip', 'sort'}, optional*) – If true the data points with NaN are removed using `numpy.isfinite()` and the data is sorted by the x values. If 'strip' is given NaNs are removed but the data isn't sorted. If 'sort' is given the data is sorted but NaNs are left in. Default is False so the data isn't changed.
- **interp_full** (*float, optional*) – This interpolates the data to give an even spacing using the inbuilt method *to_even()*. The default is None and the interpolation isn't done.

Variables

- **values** (*numpy.ndarray*) – Two column numpy array with the x and y data in.
- **x** (*numpy.ndarray*) – x data in a 1D numpy array.
- **y** (*numpy.ndarray*) – The y data in a 1D numpy array.
- **both** ((*numpy.ndarray*, *numpy.ndarray*)) – A two value tuple with the *x* and *y* in.

Notes

Mathematical operations applied to the Data class just effects the *y* values, the *x* values stay the same. To act two *Data* objects together the *x* values need to agree. *Data* objects also be mathematically acted to array_like objects (`numpy.asarray()`) of length 1 or equal to the length of the Data.

__abs__()

Calculates the absolute value of the y values.

__add__(*other*)

Addition of the y values.

__eq__(*other*)

Data class is only equal to other Data classes with the same data.

__floordiv__(*other*)

Floor division on the y values.

__getitem__(*k*)

Indexing returns a subset of the Data object.

If given a slice or and array of boolean a new Data object is produced. If given a int a length two array with [x, y] is returned.

__iter__()

The iteration happens on the values, like if was numpy array.

__mod__(*other*)

Performs the modulus with the y values.

__mul__(*other*)

Multiplication of the y values.

__neg__()

Negates the y values

__pos__()

Performs a unity operation on y values

__pow__(*other*)

Takes the power of the y values.

__setitem__(*k*, *v*)

Item assignment is not allowed in Data objects.

This kind of action is possible with the functions *set_x()*, *set_y()*, and *set_data()*.

__sub__(*other*)

Subtraction of the y values.

__truediv__(*other*)

Division of the y values.

append(*new_data*, *in_place=False*)

This adds values to the end of the data object.

Parameters

- **new_data** (*Data*) – These are the values to add onto the end of the data object
- **in_place** (*bool, optional*) – Weather to edit the object or to return a new one. The default is *False* which returns a new object.

Returns

combined_data (*Data*) – If in_place is *False* then a new Data object is returned.

apply_x(*function*)

This takes a function and applies it to the x values.

Parameters

function (*Callable*) – The function to apply to the x values.

Returns

transformed_data – Data class with new x values.

apply_y(*function*)

This takes a function and applies it to the y values.

Parameters

function (*Callable*) – The function to apply to the y values.

Returns

transformed_data – Data class with new y values.

property both

A two value tuple with the *x* and *y* in.

interp_number(*num_points*, *kind='linear'*)

Evenly interpolates in x the data for a fixed point number.

This uses *Data.interp_range()* specifying *num_points* and giving the maximum range of x points.

Parameters

- **num_points** (*int*) – The number of points to interpolate.
- **kind** (*str or int, optional*) – The type of interpolation to use. Passed to `scipy.interpolate.interp1d()`, default is *linear*.

Returns

interpolated_data (*Data*) – A Data object with evenly interpolated points.

interp_range(*min_x*, *max_x*, *step_size=None*, *num_points=None*, *shift_step=True*, *kind='linear'*)

Evenly interpolates in x the data between a min and max value.

This is used for combining datasets with corresponding but different x values. Either *step_size* or *num_points* can be defined. If *step_size* is defined `numpy.arange()` is used. If *num_points* is defined `numpy.linspace()` is used. If using *step_size* it rounds *min_x* to the next integer value of the steps, unless *shift_step* is *False*.

If values outside the range of the original data need to be passed to be interpolated, this is possible with *Data.interp_values()*. It uses `scipy.interpolate.interp1d()`.

Parameters

- **min_x** (*float*) – The minimum x value in the interpolation.
- **max_y** (*float*) – The maximum x value in the interpolation.
- **step_size** (*float, optional*) – The step size between each point. Either this or num_points must be defined.
- **num_points** (*int, optional*) – The number of points to interpolate. Either this or step_size must be defined.
- **shift_step** (*bool, optional*) – If the *min_x* value should be rounded to the next whole step. The default is True.
- **kind** (*str or int, optional*) – The type of interpolation to use. Passed to `scipy.interpolate.interp1d()`, default is *linear*.

Returns

interpolated_data (*Data*) – A Data object with evenly interpolated points.

interp_step(*step_size*, *shift_step=True*, *kind='linear'*)

Evenly interpolates in x the data between a min and max value.

This uses *Data.interp_range()* specifying *step_size* and giving the maximum range of x points. If using *step_size* it rounds *min_x* to the next integer value of the steps, unless *shift_step* is *False*.

Parameters

- **step_size** (*float, optional*) – The step size between each point. Either this or num_points must be defined.
- **shift_step** (*bool, optional*) – If the *min_x* value should be rounded to the next whole step. The default is True.
- **kind** (*str or int, optional*) – The type of interpolation to use. Passed to `scipy.interpolate.interp1d()`, default is *linear*.

Returns

interpolated_data (*Data*) – A Data object with evenly interpolated points.

interp_values(*x_vals*, *kind='linear'*, *bounds_error=True*, *fill_value=nan*, *strip_sort=False*)

Produce Data object from interpolating x values.

This uses `scipy.interpolate.interp1d()` to produce a Data object by interpolating y values from given x values.

Parameters

- **x_vals** (*array_like*) – The x values to interpolate which will be the x values.
- **kind** (*str or int, optional*) – The type of interpolation to use. Passed to `scipy.interpolate.interp1d()`, default is *linear*.
- **bounds_error** (*bool, optional*) – If default of *True* data outside the existing range will throw an error. If *False* then the value is set by *fill_value*.
- **fill_value** (float or (float, float) or *extrapolate*, optional) – If bounds_error is *False* then this value will be used outside the range. Passed to `scipy.interpolate.interp1d()`.
- **strip_sort** (*bool, optional*) – The default is *False*, where to sort and remove NaNs from the Data object before returning.

Returns

interpolated_data (*Data*) – A Data object with the given x values and interpolated y values.

max_x()

This provides the highest value of x

Returns
x_max (*float*) – The maximum value of x

min_x()

This provides the lowest value of x

Returns
x_min (*float*) – The minimum value of x

plot(**args*, *axis=None*, ***kwargs*)

Simple plotting utility

Makes use of matplotlib function matplotlib.pyplot.plot(). Runs `matplotlib.pyplot.plot(self.x, self.y, *args, **kwargs)` If provided an axis keyword which operates so that if given `axis.plot(self.x, self.y, *args, **kwargs)`.

set_data(*idx*, *val*)

This is used for setting x and y values.

Works similarly to `Data.values[idx] = val` but with more error checking. The previous code would also work (and be faster) but more care should be taken. The built in function `slice(start, end, step)()` maybe useful.

Parameters

- **idx** (*slice, int*) – Objects that can be passed to a numpy.ndarray as an index.
- **val** (*numpy.ndarray, Data*) – The values to assign to the indexed values. This can only be a two column numpy.ndarray or a *Data* object.

set_x(*idx*, *val*)

This is used for setting x values.

Works similarly to `Data.x[idx] = val` but with more error checking. The previous code would also work (and be faster) but more care should be taken. The built in function `slice(start, end, step)()` maybe useful.

Parameters

- **idx** (*slice, int*) – Objects that can be passed to a numpy.ndarray as an index.
- **val** (*numpy.ndarray*) – The values to assign to the indexed x values.

set_y(*idx*, *val*)

This is used for setting y values.

Works similarly to `Data.y[idx] = val` but with more error checking. The previous code would also work (and be faster) but more care should be taken. The built in function `slice(start, end, step)()` maybe useful.

Parameters

- **idx** (*slice, int*) – Objects that can be passed to a numpy.ndarray as an index.
- **val** (*numpy.ndarray*) – The values to assign to the indexed y values.

sort()

Sorts the data set in x and returns the new array.

Returns
sorted_data (*Data*) – A Data class with the sorted data.

`spacing_x()`

Returns the average spacing in x

Returns
x_max (*float*) – The average spacing in the x data

`strip_nan()`

This removes any row which has a nan or infinite values in.

Returns
stripped_data (*Data*) – Data class without non-finite values in.

`to_csv(filename, columns=['X', 'Y'], **kwargs)`

Saves the data as a simple csv

Uses `pandas.DataFrame.to_csv()` and kwargs are pass to it. The index keyword is set to False by default.

Parameters

- **filename** (*str*) – Filename to save the data as.
- **columns** (*[str, str]*) – The title of the two columns.

`to_even(step_size, shift_step=True, kind='linear')`

Evenly interpolates the data and updates the data object.

This uses *Data.interp_range()* specifying *step_size* and giving the maximum range of x points. If using *step_size* it rounds *min_x* to the next integer value of the steps, unless *shift_step* is *False*.

Parameters

- **step_size** (*float, optional*) – The step size between each point. Either this or num_points must be defined.
- **shift_step** (*bool, optional*) – If the *min_x* value should be rounded to the next whole step. The default is True.
- **kind** (*str or int, optional*) – The type of interpolation to use. Passed to `scipy.interpolate.interp1d()`, default is *linear*.

`property values`

Two column numpy array with the x and y data in.

`property x`

x data in a 1D numpy array.

`x_cut(x_min, x_max)`

This cuts the data to a region between x_min and x_max.

Parameters

- **x_min** (*float*) – The minimal x value to cut the data.
- **x_max** (*float*) – The maximal x value to cut the data.

Returns
cut_data (*Data*) – A data object with the values cut to the given x range.

property y

y data in a 1D numpy array.

y_from_x(*x_val*, *bounds_error=True*, *kind='linear'*)

Gives the y value for a certain x value or set of x values.

Parameters

- **x_val** (*float*) – X values to interpolate y values from.
- **bounds_error** (*bool, optional*) – If an error should thrown in x value is out of range, default True.
- **kind** (*str or int, optional*) – The type of interpolation to use. Passed to `scipy.interpolate.interp1d()`, default is *linear*.

Returns

y_val (*float or numpy.ndarray*) – Corresponding to the requested x values in an array if only one value is given a float is returned.

gigaanalysis.data.**collect_y_values**(*data_list*)

Collates the y values into a array from a collection of Data objects.

This takes either a list or dictionary of Data objects and collects the y values into one array. This can be useful of special comparisons such as trimmed means and standard deviations.

Parameters

data_list (*list or dict*) – A list of Data objects or a dictionary where the values are Data objects. The x values or all of these need to be the same.

Returns

- **x_vals** (*numpy.ndarray*) – One copy of the x values of the arrays.
- **all_data** (*numpy.ndarray*) – All the y data from the different data objects each on in it's own column.

gigaanalysis.data.**concatenate**(*data_list*, *strip_sort=False*)

Combines our collection of Data objects into one.

This takes either a list, dictionary, or tuple of Data objects or arrays and concatenates their values into one data object. This makes use of `numpy.concatenate()`.

Parameters

- **data_list** (*list or dict*) – The collection of Data objects to combine.
- **strip_sort** (*bool or {'strip', 'sort'}, optional*) – This will pass to the strip_sort keyword argument when producing the final Data object.

Returns

concatenated_data (*Data*) – The data combined into one Data object.

gigaanalysis.data.**empty_data**()

Generates an empty *Data* object.

This is useful for place holding, and takes no parameters.

Returns

empty_data (*Data*) – A Data object that contains no data points.

gigaanalysis.data.**gen_rand**(*n*, *func=None*, *seed=None*, *interp_full=None*)

Produces Data object with random values.

This uses numpy.random.Generator.random() to produce a *Data* object. The numbers in both x and y values are continually increasing in steps between 0 and 1. A function can be applied to the y values.

Parameters

- **n** (*int*) – Number of data point to have in the object.
- **func** (*function*) – A function with one parameter to transform the y values.
- **seed** (*float*) – Seed to be passed to numpy.random.default_rng()
- **interp_full** (*float, optional*) – If given the data is evenly interpolated, passed to *Data*. The default is *None* which doesn't interpolate the data.

Returns

data (*Data*) – The generated data object.

gigaanalysis.data.**interp_set**(*data_list*, *x_vals*, *kind='linear'*)

Interpolates all Data objects in list or dictionary.

This applied *Data.interp_values()* to every item in the set and returns a new set.

Parameters

- **data_list** (*list or dict*) – A list or dictionary of Data objects to interpolate.
- **x_vals** (numpy.ndarray) – The x values to interpolate to produce the new set.
- **kind** (*str or int, optional*) – The type of interpolation to use. Passed to scipy.interpolate.interp1d(), default is *linear*.

Returns

interpolated_set (*list or dict*) – The new set of Data is the same form but with interpolated values.

gigaanalysis.data.**load_dict**(*file_name*, *name_space='/'*, *strip_sort=False*, *interp_full=None*, ***kwargs*)

Loads from a file a dictionary full of Data objects.

The type of file it loads is the default produced by *save_dict()*. It assumes there is one line for the headers and they are used for the keys of the dictionary. It also removes NaNs at the end of each sweep to undo what is produced by uneven length of data objects. It makes use of pandas.read_csv(), and extra keyword arguments are passed to there.

Parameters

- **file_name** (*str*) – The name and location of the file.
- **name_space** (*str*) – The string that separates the key from the x and y names.
- **strip_sort** (*bool or {'strip', 'sort'}, optional*) – Passed to *Data*. If true the data points with NaN are removed using numpy.isfinite() and the data is sorted by the x values. If 'strip' is given NaNs are removed but the data is not sorted. If 'sort' is given the data is sorted but NaNs are left in. Default is False so the data isn't changed.
- **interp_full** (*float, optional*) – Passed to *Data*. This interpolates the data to give an even spacing using the inbuilt method to_even(). The default is None and the interpolation isn't done.

Returns

data_dict (*{str: Data}*) – The data contained in the file in the form of a dictionary where the keys are obtained from the header of the data file.

gigaanalysis.data.**match_x**(*data_list*, *step_size=None*, *num_points=None*, *shift_step=True*)

This transform a collection of dataset to have the same x values.

This applies *Data.interp_values()* to every data object with the largest possible range of x values to produce the new set of data. This is useful if the data object want to be combined arithmetically.

Parameters

- **data_list** (*list or dict of Data*) – A list of data objects or dictionary with data objects as the values.
- **step_size** (*float, optional*) – Sets the spacing in the x values to a fixed amount if given numpy.arange() is called.
- **num_points** (*int, optional*) – The number of points to generates for the x values only used if *step_size* is not given or None. If used numpy.linspace() is called.
- **shift_step** (*bool, optional*) – Only valid if step_size is not new. The default is True and then the first value is an integer number of the steps. If False the lowest x value is used as the first value of the step.

Returns

new_data_list (*list or dict of Data objects*) – The new data objects with the x values that are interoperated to be all the same. If a dict is provided a dict is returned with the same keys as before.

gigaanalysis.data.**mean**(*data_list*)

Preforms the mean of the y data a set of Data class objects.

Parameters

data_list (*[Data]*) – List of Data objects to sum together can also be a dictionary.

Returns

averaged_data (*Data*) – A Data object with the summed y values of the original data sets.

gigaanalysis.data.**save_arrays**(*array_list*, *column_names*, *file_name*, ***kwargs*)

Writes a list of arrays to csv.

This saves a collection of one dimensional numpy.ndarray stored in a list into a .csv file. It does this by passing it to a pandas.DataFrame object and using the method *to_csv*. If the arrays are different lengths the values are padded with NaNs. kwargs are passed to pandas.DataFrame.to_csv().

Parameters

- **array_list** (*[numpy.ndarray]*) – A list of 1d numpy.ndarrays to save to the .csv file.
- **columns_names** (*[str]*) – A list of column names for the .csv file the same length as the list of data arrays.
- **file_name** (*str*) – The file name to save the file as.

gigaanalysis.data.**save_data**(*data_list*, *data_names*, *file_name*, *x_name='X'*, *y_name='Y'*, *name_space='/'*, *no_sapce=True*, ***kwargs*)

Saves a list of data objects in to a .csv file.

This works by passing to *save_arrays()* and subsequently to pandas.DataFrame.to_csv(). kwargs are passed to pandas.DataFrame.to_csv()

Parameters

- **data_list** (*[gigaanalysis.data.Data]*) – A list of Data objects to be saved to a .csv file

- **data_names** (*[str]*) – A list the same length as the data list of names of each of the data objects. These will make the first half of the column name in the .csv file.
- **file_name** (*str*) – The name the file will be saved as.
- **x_name** (*str, optional*) – The string to be append to the data name to indicate the x column in the file. Default is 'X'.
- **y_name** (*str, optional*) – The string to be append to the data name to indicate the y column in the file. Default is 'Y'.
- **name_space** (*str, optional*) – The string that separates the data_name and the x or y column name in the column headers in the .csv file. The default is '/'.

gigaanalysis.data.**save_dict**(*data_dict*, *file_name*, *x_name='X'*, *y_name='Y'*, *name_space='/'*, ***kwargs*)

Saves a dictionary of data objects in to a .csv file.

This works by passing to *save_data()* and subsequently to `pandas.DataFrame.to_csv()`. The names of the data objects are taken from the keys of the data_dict. kwargs are passed to `pandas.DataFrame.to_csv()`

Parameters

- **data_list** (*[gigaanalysis.data.Data]*) – A dictionary of Data objects to be saved to a .csv file. The keys of the dictionary will be used as the data names when passed to *save_data()*.
- **file_name** (*str*) – The name the file will be saved as.
- **x_name** (*str, optional*) – The string to be append to the data name to indicate the x column in the file. Default is 'X'.
- **y_name** (*str, optional*) – The string to be append to the data name to indicate the y column in the file. Default is 'Y'.
- **name_space** (*str, optional*) – The string that separates the data_name and the x or y column name in the column headers in the .csv file. The default is '/'.

gigaanalysis.data.**sum**(*data_list*)

Preforms the sum of the y data a set of Data class objects.

Parameters
data_list (*[Data]*) – List of Data objects to sum together.

Returns
summed_data (*Data*) – A Data object with the summed y values of the original data sets.

gigaanalysis.data.**swap_xy**(*data*, ***kwargs*)

Interchange the independent and dependent variables.

This takes a *Data* object and returns a new one with the x and y variables swapped around. Keyword arguments are pass to the *Data* class.

Parameters
data (*Data*) – The data to switch the x and y values.

Returns
swapped_data (*Data*) – A new *Data* object with x and y values switched.

gigaanalysis.data.**y_from_fit**(*data*, *x_value*, *x_range*, *poly_deg=1*, *as_Data=False*, *std=False*)

Fits a polynomial over a range to interpolate a given value.

This makes use of `numpy.polyfit()` to find an interpolated value of y form a data object and a given x value.

Parameters

- **data** (*Data*) – The data to interpolate the value from. Should be a sorted data object.
- **x_value** (*float or numpy.ndarray*) – The value of the independent to obtain the associated dependent variable.
- **x_range** (*float*) – The range of independent variables to perform the fit over.
- **poly_deg** (*int, optional*) – The order of the polynomial to use when fitting to find the result. The default is *1* which is a linear fit.
- **as_Data** (*bool, optional*) – If default of False y values are given as an float or an array. If *True* then a Data object is returned.
- **std** (*bool, optional*) – If *fit* or 'residual' then the standard deviation is returned after the values. The standard deviation can either be calculated from the error in the fit (using 'fit') or from the distribution of the residuals of the fit (using 'residual'). The default value is *False*, where only the value will be returned.

Returns

y_value (*float, numpy.ndarray, or Data*) – The y values obtained at the associated value of x for the fit performed. The type depends if multiple points are requested and if 'as_Data` is set. If *std* is *True* then the standard deviation is followed in the same format.

2.3 GigaAnalysis - Mathematics Functions - gigaanalysis.mfunc

This module contains a large collections of simple functions that are for performing basic maths using *Data* objects. The functions are currently in three types. There are the basic functions that act on the dependent variables and return a new *Data* object. There is the make functions that produce data in a certain form given parameters and the x values. There are also a few more functions to do with FFTs, differentiation and integration. **ufunc Functions**

Here is the collection of simple functions that applies numpy ufuncs to the *Data* objects. These all work the same and are based around the function *apply_func()*. This functionality can be achieved with methods on the *Data* class but this was added to provide more readability.

gigaanalysis.mfunc_ufunc.**abs**(*data*, *act_x=False*, *as_Data=True*)

Applies `numpy.abs()` using *apply_func()*.

gigaanalysis.mfunc_ufunc.**apply_func**(*data*, *func*, *act_x=False*, *as_Data=True*)

Applies numpy ufuncs to GigaAnalysis Data objects.

This applies the function to the relevant variable and returns the object in the format specified. The methods *Data.apply_x()* and *Data.apply_y()* perform a very similar function.

Parameters

- **data** (*Data*) – The Data object to act on with the function.
- **func** (*numpy.ufunc*) – The function to act with on the data.
- **act_x** (*bool, optional*) – If default of *False* act on the y values, otherwise if *True* act on the x values.
- **as_Data** (*bool, optional*) – If the default of *True* returns a *Data* object, otherwise if *False* returns a `numpy.ndarray` of the values acted on instead.

Returns

out_data (*Data, numpy.ndarray*) – The data with the chosen variable acted on, and returned either as a Data object with the other variable unchanged or as a `numpy.ndarray` without the data from the other variable.

gigaanalysis.mfunc_ufunc.**arccos**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.arccos() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**arccosh**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.arccosh() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**arcsin**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.arcsin() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**arcsinh**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.arcsinh() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**arctan**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.arctan() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**arctanh**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.arctanh() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**cos**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.cos() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**cosh**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.cosh() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**exp**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.exp() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**exp10**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.exp10() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**exp2**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.exp2() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**log**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.log() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**log10**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.log10() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**log2**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.log2() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**reciprocal**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.reciprocal() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**sin**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.sin() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**sinh**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.sinh() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**sqrt**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.sqrt() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**square**(*data*, *act_x=False*, *as_Data=True*)
Applies numpy.square() using *apply_func()*.

gigaanalysis.mfunc_ufunc.**tan**(*data*, *act_x=False*, *as_Data=True*)

Applies `numpy.tan()` using *apply_func()*.

gigaanalysis.mfunc_ufunc.**tanh**(*data*, *act_x=False*, *as_Data=True*)

Applies `numpy.tanh()` using *apply_func()*.

Make Functions

Here are a few functions that are for producing mathematical functions of a certain form. These are also used by the plotting functions in the module *fit*. Other functions more specific to the certain areas of physics are included in the relevant modules.

gigaanalysis.mfunc_make.**make_gaussian**(*x_data*, *amp*, *mean*, *std*, *offset=0*, *as_Data=True*)

This function generates Gaussian functions

The form of the equation is `amp*np.exp(-0.5*np.power((x_data - mean)/std, 2)) + offset` The offset is a keyword argument that doesn't need to be applied. The amplitude refers the the maximum value at the top of the peak.

Parameters

- **x_data** (*numpy.ndarray*) – The values to compute the y values of.
- **amp** (*float*) – The maxiumal value of the Gaussian function.
- **mean** (*float*) – The centre of the Gaussian function.
- **std** (*float*) – The width of the Gaussian given as the standard deviation in the same units as the *x_data*.
- **offset** (*float, optional*) – Shift all of the y values by a certain amount, default is for no offset.
- **as_Data** (*bool, optional*) – If the default of *True* returns a *Data* object with the x values given and the cosponsoring y values. If False returns a class:*numpy.ndarray*.

Returns

results (*Data or numpy.ndarray*) – The values expected from the Gaussian with the given parameters

gigaanalysis.mfunc_make.**make_poly**(*x_data*, **p_vals*, *as_Data=True*)

Generates a polynomial from the coefficients.

The point of this function is to generate the values expected from a linear fit. It is designed to take the values obtained from `numpy.polyfit()`. For a set of p_vals of length n+1 `y_data = p_vals[0]*x_data**n + p_vals[0]*x_data**(n-1) + ... + p_vals[n]`

Parameters

- **x_data** (*numpy.ndarray*) – The values to compute the y values of.
- **p_vals** (*float*) – These are a series of floats that are the coefficients of the polynomial starting with with the highest power.
- **as_Data** (*bool, optional*) – If the default of *True* returns a *Data* object with the x values given and the cosponsoring y values. If False returns a class:*numpy.ndarray*.

Returns

results (*Data or numpy.ndarray*) – The values expected from a polynomial with the specified coefficients.

gigaanalysis.mfunc_make.**make_sin**(*x_data*, *amp*, *wl*, *phase*, *offset=0*, *as_Data=True*)

This function generates sinusoidal functions.

The form of the equation is `amp*np.sin(x_data*np.pi*2./wl + phase*np.pi/180.) + offset` The offset is a keyword argument that doesn't need to be applied.

Parameters

- **x_data** (*numpy.ndarray*) – The values to compute the y values of.
- **amp** (*float*) – Amplitude of the sin wave.
- **wl** (*float*) – Wavelength of the sin wave units the same as *x_data*.
- **phase** (*float*) – Phase shift of the sin wave in degrees.
- **offset** (*float, optional*) – Shift all of the y values by a certain amount, default is for no offset.
- **as_Data** (*bool, optional*) – If the default of *True* returns a *Data* object with the x values given and the cosponsoring y values. If False returns a class:*numpy.ndarray*.

Returns

results (*Data or numpy.ndarray*) – The values expected from the sinusoidal with the given parameters

FFT Functions

This module contains a large collections of simple functions that are for performing basic maths using *Data* objects. These are for performing a Fast Fourier Transform (FFT) mostly using the function *fft()*. There are also some functions for identifying the FFT peaks.

gigaanalysis.mfunc_fft.**fft**(*data*, *n=65536*, *window='hann'*, *freq_cut=0.0*)

Performs an Fast Fourier Transform on the given data.

This assumes that the data is real, and the data provided needs to be evenly spaced. Makes use of `numpy.fft.rfft()`. This takes into account the x values to provide the frequencies in the correct units.

Parameters

- **data** (Data) – The data to be FFT. This must be evenly spaced in x.
- **n** (*int, optional*) – The number of points to make the FFT extra points will be zero padded. The default is `2**16 = 65536`. If the data is longer than the value of n, n is rounded up to the next power of 2.
- **window** (*str, optional*) – The type of windowing function to use taken from `scipy.signal.get_window()`. The default is 'hann'.
- **freq_cut** (*float, optional*) – The frequency to drop all the higher from. The default is 0 which means that all the frequencies are kept.

Returns

fft_result (*Data*) – A data object with the FFT frequencies in the x values and the amplitudes in the y values.

gigaanalysis.mfunc_fft.**get_peaks**(*data*, *n_peaks=4*, *as_Data=False*, ***kwargs*)

This returns the peaks in a data object as a numpy array.

Using `scipy.signal.find_peaks()` the peaks in the data object are found. `**kwargs` are passed to that function. This is most commonly used when examining FFT data.

Parameters

- **data** (Data) – The data to look for peaks in.

- **n_peaks** (*int, optional*) – The number of peaks to return, the default is 4.
- **as_Data** (*bool, optional*) – If *True* the peak info is returned as a *Data* object. The default is *False*.

Returns

peak_info (*numpy.ndarray*) – A two column numpy array with the with the location and the amplitude of each peak.

gigaanalysis.mfunc_fft.**peak_height**(*data*, *position*, *x_range*, *x_value=False*)

Gives the info on highest peak in a given region.

It achieves this my trimming the data to a region and then returning the maximum y_value. This is useful for extracting peak heights from an FFT.

Parameters

- **data** (*Data*) – The data to extract the peak hight from.
- **position** (*float*) – The central position to search for the peak.
- **x_range** (*float*) – The range in x to look for the peak. This is the total range so will extend half of this from `position`.
- **x_value** (*bool, optional*) – If *True* the x value and the y value is returned. The default is *False* which only returns the y value.

Returns

- **x_peak** (*float*) – If x_value is *True* the x position of the peak is returned.
- **y_peak** (*float*) – The y_value of the peak. Which is the highest y value in a given range of the data.

Transformation Functions

These are all functions that are applied to a *Data* class and return a similar object. These are either simple filters, transforms, or differentiation or intergeneration functions.

gigaanalysis.mfunc_trans.**deriv**(*data*, *x_window*, *deriv_order*, *polyorder=None*, ***kwargs*)

This applies a LOESS filter to the data.

The LOESS filter is applied using `scipy.signal.savgol_filter()`. This fits a polynomial to many sections of the data and uses the central value as the value of the transformed data. Keyword arguments can be given and will be passed to `scipy.signal.savgol_filter()`.

Parameters

- **data_set** (*Data*) – The data to be smoothed. The points must be evenly spaced in x for this to be applied.
- **x_window** (*float*) – The length of the window to apply in the same units as the x values.
- **polyorder** (*int*) – The order of the polynomial to apply.

Returns

smoothed_data (*Data*) – A data object after the data has been smoothed with LOESS filter.

gigaanalysis.mfunc_trans.**integrate**(*data*)

Integrate a *Data* object cumulatively.

This uses `scipy.integrate.cumulative_trapezoid()` to preform a cumulative integration over the data set. This will sort the data set in the process.

Parameters
data (Data) – The *Data* object to preform the integration on.

Returns
integral (*Data*) – The integral the of the data and will be the same size.

gigaanalysis.mfunc_trans.**invert_x**(*data*)

Inverts the x values and re-interpolates them.

This is useful for quantum oscillations because the periodicity in inverse magnetic field.

Parameters
data (Data) – The data to invert

Returns
inverted_data (*Data*) – A data object with points evenly separated in the new inverted x values.

gigaanalysis.mfunc_trans.**loess**(*data*, *x_window*, *polyorder*, ***kwargs*)

This applies a LOESS filter to the data.

The LOESS filter is applied using scipy.signal.savgol_filter(). This fits a polynomial to many sections of the data and uses the central value as the value of the transformed data. Keyword arguments can be given and will be passed to scipy.signal.savgol_filter().

Parameters

- **data_set** (Data) – The data to be smoothed. The points must be evenly spaced in x for this to be applied.
- **x_window** (*float*) – The length of the window to apply in the same units as the x values.
- **polyorder** (*int*) – The order of the polynomial to apply.

Returns
smoothed_data (*Data*) – A data object after the data has been smoothed with LOESS filter.

gigaanalysis.mfunc_trans.**poly_reg**(*data*, *polyorder*, *sigma=None*)

This applied a polynomial fit to data and returns the fit curve.

This uses numpy.polyfit() and filters the data down to a simple polynomial. This can be used for subtracting polynomial background from data sets.

Parameters

- **data** (Data) – The data to apply the fit too.
- **polyorder** (*int*) – The order of the polynomial to use in the fit.
- **sigma** (*numpy.ndarray, optional*) – The standard divation of the points of the data. Needs to be a numpy.ndarray the same length as the data. Default is None where every point is evenly weighted.

Returns
poly_smoothed_data (*Data*) – The polynomial which was the best fit to the data with the original x values and the generated values.

2.4 GigaAnalysis - Data Set Management - gigaanalysis.dset

This module has functions to save nested dictionaries with Data objects as the values. It provides the functionality to save and read from HDF5 files using h5py and also .csv files. It also can create and use pandas.DataFrame to store and display associated meta data.

gigaanalysis.dset.**array_from_hdf5**(*file_name*, *location*)

This reads a dataset in a HDF5 file to a numpy array.

This function is to read the data saved using the *array_to_hdf5()*. It reads the data and the attributes using h5py.File and returns the result.

Parameters

- **file_name** (*str*) – The name of the HDF5 file to be read.
- **location** (*str*) – The location of the dataset with the groups and dataset name separated by "/".

Returns

- **data** (*numpy.ndarray*) – A numpy array containing the data in the data set.
- **attributes** (*dict*) – A dictionary containing the attributes of the data set that was read. If there was no attributes then the dictionary will be empty.

gigaanalysis.dset.**array_to_hdf5**(*data*, *file_name*, *location*, *attributes=None*, *overwrite=False*)

Saves a numpy array to a HDF5 file.

This is for saving a plane numpy.ndarray to a HDF5 using h5py.File. This is meant to work in the same style as *set_to_hdf5()*. It also can save a set of attributes in the form of a dictionary.

Parameters

- **data** (*numpy.ndarray*) – The data to save to the file in a numpy array.
- **file_name** (*str*) – The name of the HDF5 file to save the data to.
- **location** (*str*) – The location of the h5py.Dataset, which is a string with the groups and the data set name separated by "/".
- **attributes** (*dict of {str: val}, optional*) – A dictionary of meta data to attach to the data set. The keys of the dictionary need to be *str*. Default is None and attaches no attributes to the data set.
- **overwrite** (*bool, optional*) – If default of *False* the existing file is not overwritten and is instead added to. This will throw an error if trying to save to a location of an already existing dataset.

gigaanalysis.dset.**check_set**(*data_set*, *meta_df=None*, *higher_key=()*)

Checks the data_set and metadata data frame is the correct from.

This goes through the nested dictionaries and checks that the values contained are either dict or *gigaanalysis.data.Data* objects. If objects other than these are found errors are thrown. The metadata dictionary meta_df is checked that every Data has a row that is describing it in the meta_df.

Parameters

- **data_set** (*dict of {str: dict or Data}*) – A dictionary containing either nested dictionaries or *gigaanalysis.data.Data* objects.

- **meta_df** (*pandas.DataFrame*) – Metadata held in a pandas.DataFrame where the indexes are a the keys of data_set and the columns provide information about the Data objects. For nested dictionaries hierarchical indexing is used (pandas.MultiIndex).
- **higher_key** (*tuple, optional*) – Tuple with keys in used for the regression to start in a nested dictionary.

Returns

count (*int*) – The number of layers of the data_set.

gigaanalysis.dset.**print_hdf5**(*file_name*)

Prints the names and attributes of the contents of a HDF5 file.

Parameters

file_name (*str*) – The name of the HDF5 file to read.

gigaanalysis.dset.**set_from_hdf5**(*file_name*, *location='/'*)

Reads a HDF5 file and returns a dataset and a metadata table.

This reads a HDF5 file using h5py.File, and produces a dataset comprising of a nested dict which contains *gigaanalysis.data.Data* objects. The dataset is accompanied by a metadata table in the form of a pandas.DataFrame with the indexes are the same as the keys of the dictionaries.

Parameters

- **file_name** (*str*) – The name of the HDF5 file to read.
- **location** (*str, optional*) – The location of the group in the HDF5 file which contains the dataset to be read. The default is the root group.

Returns

- **data_set** (*dict of {str: dict or Data}*) – A dictionary containing either nested dictionaries or *gigaanalysis.data.Data* objects.
- **meta_df** (*pandas.DataFrame, optional*) – Metadata held in a pandas.DataFrame where the indexes are a the keys of data_set and the columns provide information about the Data objects. For nested dictionaries hierarchical indexing is used (pandas.MultiIndex).

gigaanalysis.dset.**set_to_hdf5**(*data_set*, *file_name*, *meta_df=None*, *location='/'*, *overwrite=False*, *info_attr=None*)

This saves a data set to a HDF5 file.

This saves a data set of made of nested dict of *gigaanalysis.data.Data* to a HDF5 file, using h5py.File. This can also take a pandas.DataFrame containing the associated meta_data.

Parameters

- **data_set** (*dict of {str: dict or Data}*) – A dictionary containing either nested dictionaries or *gigaanalysis.data.Data* objects.
- **file_name** (*str*) – The file name to save the HDF5 file with
- **meta_df** (*pandas.DataFrame, optional*) – Metadata held in a pandas.DataFrame where the indexes are a the keys of the *data_set* dict and the columns provide information about the Data objects. For nested dictionaries hierarchical indexing is used (pandas.MultiIndex).
- **location** (*str, optional*) – The location of the HDF5 group to save the data to. The default is the root group.
- **overwrite** (*bool, optional*) – If the function should overwrite existing HDF5 file. The default is to not overwrite.

- **info_attr** (*str, optional*) – If a string is given this is set as an HDF5 attribute to group. This can hold a description of data if required.

gigaanalysis.dset.**sort_dset**(*dataset*, *apply_key=None*, *sort_key=None*, *check_data=True*)

This sorts and formats the keys in a dataset.

This is useful after loading a dataset from a HDF5 file, as keys that were floats will have been set to strings and then loaded by the leading digit. This function can apply a function to each key and then sort them.

Parameters

- **dataset** (*recursive dict of Data*) – This is the dataset to sort which are nested dictionaries of Data objects.
- **apply_key** (*function or list of function, optional*) – This is a function that will be applied to the keys to reformat them before the are reordered. If a list is given then each function will be applied on each layer of the dataset in turn. If *None* then no function is applied. The default is *None*.
- **sort_key** (*function or list of function, optional*) – This is the key that is passed to *sorted* to sort the dataset based on its keys. If a list is given then each function will be applied to each layer of the dataset. If *None* then no key is passed and the default sorting behaviour is used. If the string 'pass' is given then no sorting is applied. The default is *None*.
- **check_data** (*bool, optional*) – Whether to check if the objects in the dict are Data objects. The default is True.

Returns

dataset (*recursive dict of Data*) – The dataset after the functions have been applied to the keys and the keys and then they have been sorted.

2.5 GigaAnalysis - Fitting - gigaanalysis.fit

This module contains the *Fit_result* class which is used for fitting functions to the GigaAnalysis Data objects. It also contains some common expressions that are needed as well as functions that fit a Data class with them.

class gigaanalysis.fit.**Fit_result**(*func*, *popt*, *pcov*, *results*, *residuals*)

Bases: object

This class is to hold the results of the fits on data objects.

Parameters

- **func** (*function*) – The function used in the fitting.
- **popt** (*numpy.ndarray*) – The optimum values for the parameters.
- **pcov** (*numpy.ndarray*) – The estimated covariance.
- **results** (Data) – The optimal values obtained from the fit, will be none if *full* `= `*False* when performing the fit.
- **residuals** (Data) – The residuals of the fit, will be none if *full* `= `*False* when performing the fit.

Variables

- **func** (*function*) – The function used in the fitting.
- **popt** (*numpy.ndarray*) – The optimum values for the parameters.

- **pcov** (*numpy.ndarray*) – The estimated covariance.
- **pstd** (*numpy.ndarray*) – The estimated standard deviation
- **results** (gigaanalysis.data.Data) – The optimal values obtained from the fit, will be none if *full* `= `*False* when performing the fit.
- **residuals** (gigaanalysis.data.Data) – The residuals of the fit, will be none if *full* `= `*False* when performing the fit.

predict(*x_vals*)

This takes a value or an array of x_values and calculates the predicated y_vales.

Parameters
x_vals (*numpy.ndarray*) – An array of x_vales.

Returns
y_vals (*Data*) – An Data object with the predicted y_values.

sample_parameters(*size*, ***kwargs*)

This samples values of the parameters from a multivariate normal distribution using the fitted values and variance.

This uses the function numpy.random.multivariate_normal() and keyword arguments are passed to it.

Parameters
size (*int*) – The number of samples to return. (More complicated behaviour is possible, see: numpy.random.multivariate_normal())

Returns
samples (*numpy.ndarray*) – A numpy array in the shape *(s, n)* where *s* is the number of samples and *n* is the number of parameters.

gigaanalysis.fit.**curve_fit**(*data*, *func*, *p0*, *full=True*, ***kwargs*)

Fit curves to Data objects with functions that produce Data objects.

This is an implementation of scipy.optimize.curve_fit() for acting on *Data* objects. This performs a least squares fit to the data of a function.

Parameters

- **data** (Data) – The data to perform the fit on.
- **func** (*function*) – The model function to fit. It must take the x values as the first argument and the parameters to fit as separate remaining arguments. It must also return a *Data* object.
- **p0** (*numpy.ndarray*) – Initial guess for the parameters. Is passed to scipy.optimize.curve_fit() included so it can be addressed positionally.
- **full** (*bool, optional*) – If *True*, *fit_result* will include residuals, and if *False* they will not be calculated and only results included.
- **kwargs** – Keyword arguments are passed to scipy.optimize.curve_fit().

Returns
fit_result (*Fit_result*) – A GigaAnalysis Fit_result object containing the results.

gigaanalysis.fit.**curve_fit_y**(*data*, *func*, *p0=None*, *full=True*, ***kwargs*)

Fit curves to Data objects with functions that produce y values.

This is an implementation of scipy.optimize.curve_fit() for acting on *Data* objects. This performs a least squares fit to the data of a function.

Parameters

- **data** (Data) – The data to perform the fit on.
- **func** (*function*) – The model function to fit. It must take the x values as the first argument and the parameters to fit as separate remaining arguments.
- **p0** (*numpy.ndarray, optional*) – Initial guess for the parameters. Is passed to scipy.optimize.curve_fit() included so it can be addressed positionally. If *None* unity will be used for every parameter.
- **full** (*bool, optional*) – If *True*, *fit_result* will include residuals, and if *False* they will not be calculated and only results included.
- **kwargs** – Keyword arguments are passed to scipy.optimize.curve_fit().

Returns

fit_result (*Fit_result*) – A GigaAnalysis Fit_result object containing the results.

gigaanalysis.fit.**gaussian_fit**(*data*, *p0=None*, *offset=False*, *full=True*, ***kwargs*)

Fits a Gaussian to a given Data object.

This uses mfunc.make_gaussian() and has the option to either include an offset from zero or not to. It then uses *curve_fit_y()* that makes use of scipy.optermize.curve_fit() which *kwargs* are passed to.

Parameters

- **data** (Data) – The Data to fit the sinusoid to.
- **p0** (*numpy.ndarray, optional*) – The initial values to begin the optimisation from. These are in order, the amplitude of the Gaussian, the central point, the standard deviation, and if included the offset from zero.
- **offset** (*bool, optional*) – If *True* an offset from zero is also included in the fit. The default is *False*.
- **full** (*bool, optional*) – If *True*, which is the default, the results and residuals are included in the returned *Fit_result*.
- **kwargs** – The keyword arguments are passed to scipy.optermize.curve_fit().

Returns

fit_result, Fit_result – A GigaAnalysis Fit_result object containing the results where the fit parameters are same as specified in mfunc.make_gaussian().

gigaanalysis.fit.**poly_fit**(*data*, *order*, *full=True*)

Fit a polynomial of a certain order to a given data set.

It uses numpy.polyfit() for the fitting. The function which is to produce the data is mfunc.make_poly().

Parameters

- **data** (Data) – The data set to perform the fit on.
- **order** (*int*) – The order of the polynomial.
- **full** (*bool, optional*) – If True fit_result will include residuals if False they will not be calculated and only results included.

Returns

fit_result (*Fit_result*) – A GigaAnalysis Fit_result object containing the results where the fit parameters are the coefficients of the polynomial. Follows the form of gigaanalysis.fit.any_poly().

gigaanalysis.fit.**sin_fit**(*data*, *p0=None*, *offset=False*, *full=True*, ***kwargs*)

Fits a sinusoid to a given Data object.

This uses `mfunc.make_sin()` and has the option to either include an offset from zero or not to. It then uses *curve_fit_y()* that makes use of `scipy.optermize.curve_fit()` which *kwargs* are passed to.

Parameters

- **data** (*Data*) – The Data to fit the sinusoid to.
- **p0** (*numpy.ndarray, optional*) – The initial values to begin the optimisation from. These are in order, the amplitude of the sin, the wavelength, the phase, and if included the offset from zero.
- **offset** (*bool, optional*) – If *True* an offset from zero is also included in the fit. The default is *False*.
- **full** (*bool, optional*) – If *True*, which is the default, the results and residuals are included in the returned *Fit_result*.
- **kwargs** – The keyword arguments are passed to `scipy.optermize.curve_fit()`.

Returns

fit_result, Fit_result – A GigaAnalysis Fit_result object containing the results where the fit parameters are same as specified in `mfunc.make_sin()`.

2.6 GigaAnalysis - Parsing - `gigaanalysis.parse`

This module contains functions for parsing datasets. Now it includes functions for identifying measurements clustered in groups and taking the average of them. This can be useful for plotting datasets from instruments that take multiple measurements at each point in a sweep.

gigaanalysis.parse.**cluster_group**(*data*, *normalise='constant'*, *threshold=None*, *relative_threshold=False*)

This identifies clusters of points close together and produces an array with each of these points indexed by their cluster.

Parameters

- **data** (*numpy.ndarray*) – The values to check if they clustered
- **normalise** (*{'constant', 'value', 'log'} optional*) – This normalises the difference between the values of the data set to better identify the clusters. 'constant' dose not perform any normalisation. 'value' divides the difference by the first value. 'log' takes the log of all the data values before preforming the difference.
- **threshold** (*float optional*) – The value the difference needs to exceed to be considered a new cluster. If no value is given then the average of the differences are used. If :param:relative_threshold is True this value is multiplied by the averages of the differences.
- **relative_threshold** (*bool optional*) – If True the given threshold is multiplied by the averages of the differences. The default is False.

Returns

groups (*numpy.ndarray*) – A numpy.ndarray the same length as the dataset containing the indexes of the groups each datum corresponds to.

gigaanalysis.parse.**end_of_dataset**(*data_set*, *minimum=True*, *look_up=None*, *interp_step=None*, *loess_window=None*, *loess_poly=2*)

Produces a Data object from the x value extent of a dataset.

This produces a Data object where each datum is composed of a value made from the key of the dataset, and the other is either the minimum or maximum x value. This is use for finding the x extent of two dimensional maps.

Parameters

- **data_set** (*dict of Data*) – The dataset to obtain the values from.
- **minimum** (*bool, optional*) – If to take the minimum or maximum x values from the dataset. The default value if *True* and this takes the minimum.
- **look_up** (*dict, optional*) – A dictionary that converts the keys from the dataset into floats to be returned. The default is *None* where the values of the keys themselves is used.
- **interp_step** (*float, optional*) – This applies the method *Data.interp_step()* to the data object after it is produced. This is useful as the data needs to be evenly interpreted before it can be smoothed. The default is *None* which does not apply the method.
- **loess_window** (*float, optional*) – This can be used to smooth the data. The default is *None* where no smoothing is applied. The value sets the range to be used for the loess window in `mfunc.loess()`. This is useful for smoothly masking the bottom of contour maps.
- **loess_poly** (*int, optional*) – The default is *2*. This this is the order of the polynomial to be used by the loess function `mfunc.loess()`.

Returns

end_data (*Data*) – The data object which is composed of the minimum or maximum x values in the dataset.

gigaanalysis.parse.**group_average**(*data*, *groups*, *error=False*, *std_factor=True*, *not_individual=False*)

This takes a set of data that has a corresponding indexed groups and produces a new set of the averages of those groups. This can also produce a corresponding set with the standard deviation of the groups.

Parameters

- **data** (*numpy.ndarray*) – The data set to preform the averages of the groups on.
- **groups** (*numpy.ndarray*) – The array with the corresponding index of the groups. Is required to be the same size as the data array.
- **error** (*bool, optional*) – Whether to produce the array of the standard deviations. Default is False
- **std_factor** (*bool, optional*) – If True which is default will output the expectation value of the standard deviation. If False will only output the standard deviation. If a group has one datum the standard deviation is given as 0 as opposed to infinite.
- **not_individual** (*bool optional*) – If True and if error is True the groups with only one datum will be dropped.

Returns

- **averages** (*numpy.ndarray*) – An array the length of the number of groups with the average of the values in the data array for the data points in each group.
- **errors** (*numpy.ndarray*) – If :param:error is True errors are returned. An array the length of the number of groups with the standard deviation of the values in the data array for the data points in each group.

gigaanalysis.parse.**read_wpd**(*file_name*, *parse_keys=None*, *sort_keys=False*, *strip_sort=True*)

Read Web Plot Digitizer output csv files.

Web Plot Digitizer is a program that can extract the data from images of scientific figures. When the program exports the data as a csv file it is in a certain format, which this function reads. The output is a gigaanaylsis dataset with the names of the web plot digitizer datasets as the keys.

Parameters

- **file_name** (*str*) – The location of csv file that Web Plot Digitizer produced.
- **parse_keys** (*callable, optional*) – If a function is given the keys are passed to it and the output is used as the new key.
- **sort_keys** (*bool, optional*) – If *True* then the keys are sorted, the default is *False* where they will be in the order in the csv file.
- **strip_sort** (*bool, optional*) – If default of *True* the option of the same name is given for each of the *Data* objects in the set.

Returns

dataset (*dict of Data*) – A dictionary where the values are *Data* objects containing the data in the csv file.

gigaanalysis.parse.**roll_dataset**(*independent*, *dependent*, *variable*, *look_up=None*, *strip_sort=True*, *drop_empty=False*)

This packs data from three arrays into a dataset.

This takes three one dimensional numpy.ndarray and uses the last one to group the first two into data objects. The first array is used to for the independent variable and the second is used for the independent variable. A dictionary can also be provided as a look up to change the dataset keys.

Parameters

- **independent** (*numpy.ndarray*) – The x values to form all the *Data* objects.
- **dependent** (*numpy.ndarray*) – The y values to form all the *Data* objects.
- **variable** (*numpy.ndarray*) – The corresponding values to group the values to the different Data objects to form the data_set.
- **look_up** (*dict or pandas.Series, optional*) – This is a dictionary that converts the values in the variable array into keys that will be used in the dictionary. The default behaviour uses the values in the variable for the keys.
- **strip_sort** (*bool, optional*) – This is *True* by default and is passed to the *strip_sort* argument of the Data when they are produced.
- **drop_empty** (*bool, optional*) – This is *False* by default and if *True* Data objects are removed if they contain no data points. This would happen if all the values retrieved were NaNs and then strip_sort was applied.

Returns

data_set (*dict of Data*) – The data set produced by combining the three data sets.

gigaanalysis.parse.**unroll_dataset**(*data_set*, *look_up=None*)

This unpacks all the values in a data set into 3 arrays.

This splits the data from a data set into three, the x and y values and the values from the key. To covert the keys into something useful a dict can be provided as a look up table.

Parameters

- **data_set** (*dict of Data*) – The data set to unroll all the values from.

- **look_up** (*dict or pandas.Series, optional*) – This is a dictionary that converts the keys in the data_set into something to place in the variable array.

Returns

- **independent** (*numpy.ndarray*) – The x values from all the *Data* objects.
- **dependent** (*numpy.ndarray*) – The y values from all the *Data* objects.
- **variable** (*numpy.ndarray*) – The corresponding keys from the data_set or values produced from passing them into the look up dictionary.

2.7 GigaAnalysis - Quantum Oscillations - gigaanalysis.qo

Here is a set of functions and classes that are useful for analysing quantum oscillation data. The general form that I assume when processing magnetic field sweeps to look for quantum oscillation are performing a background subtraction and then Fourier transforming that inverse field.

class gigaanalysis.qo.**QO**(*data*, *min_field*, *max_field*, *subtract_func*, *step_size=None*, *fft_cut=0*, *strip_nan=False*)

Bases: object

Quantum Oscillation object

This takes a sweep in magnetic field and analyses the sweep to check the presence and properties of quantum oscillations. It uses the *Data* objects to hold the information.

This class has an arbitrary subtraction function which is used to remove the background signal and leave the quantum oscillations. Other ready made classes exist that have a certain subtraction function incorporated. This class functions as their parent.

The analysis happens in 4 stages and the data is assessable at each stage as a *Data* attribute. The first stage interpolates the data evenly across the field window of interest. The second stage performs the background subtraction. The third stage inverts the data in in field. The final stage Fourier transforms the inverted signal.

Parameters

- **data** (*Data*) – The raw data of the field sweep to look for quantum oscillations in.
- **min_field** (*float*) – The lowest field value of the field range to inspect.
- **max_field** (*float*) – The highest field value of the field range to inspect.
- **subtract_func** (*calculable*) – This should take one *Data* object and return one *Data* of the same length. The input data is the interpolated sweep, and the output should be the data after the background has been subtracted.
- **step_size** (*float, optional*) – The size of field steps to interpolate. The default is 4 times the average step size in the raw data.
- **fft_cut** (*float, optional*) – The maximum frequency to consider in FFT, higher frequencies are dropped. THe default is to keep all the data.
- **strip_nan** (*bool, optional*) – If *True* non finite values are removed from the raw data. The default is *False* and this will raise an error is non finite values are in the raw data.

Variables

- **raw** (*Data*) – The data originally given to the class.
- **interp** (*Data*) – The sweep cut to the field range and interpolated.

- **sub** (Data) – The sweep after the background subtraction.
- **invert** (Data) – The subtracted sweep after inverting the field values.
- **fft** (Data) – The Fourier transform of the inverted signal.
- **min_field** (*float*) – The minimum field in the range in consideration.
- **max_field** (*float*) – The maximum field in the range in consideration.
- **step_size** (*float*) – The steps in field calculated in the interpolation.

FFT_again(*n=65536*, *window='hann'*, *freq_cut=0*)

Recalculates the FFT.

After recalcuating the FFT the new FFT is returned and also the new FFT is saved to the `fft` attribute. This makes use of `mfunc.fft()`.

Parameters

- **n** (*int, optional*) – The number of points to make the FFT extra points will be zero padded. The default is `2**16 = 65536`. If the data is longer than the value of n, n is rounded up to the next power of 2.
- **window** (*str, optional*) – The type of windowing function to use taken from `scipy.signal.get_window()`. The default is 'hann'.
- **freq_cut** (*float, optional*) – The frequency to drop all the higher from. The default is 0 which means that all the frequencies are kept.

Returns

fft_result (*Data*) – A data object with the FFT frequencies in the x values and the amplitudes in the y values.

peak_height(*position*, *x_range*, *x_value=False*)

Provides the hight of the highest FFT in a given range.

Makes use of the function `mfunc.peak_height()`.

Parameters

- **position** (*float*) – The central position to search for the peak.
- **x_range** (*float*) – The range in x to look for the peak. This is the total range so will extend half of this from `position`.
- **x_value** (*bool, optional*) – If *True* the x value and the y value is returned. The default is *False* which only returns the y value.

Returns

- **x_peak** (*float*) – If x_value is *True* the x position of the peak is returned.
- **y_peak** (*float*) – The y_value of the peak. Which is the highest y value in a given range of the data.

peaks(*n_peaks=4*, *as_Data=False*, ***kwargs*)

Finds the largest Fourier Transform peaks.

This makes use of `mfunc.get_peaks()` and the `**kwargs` are passed to `scipy.signal.find_peaks()`.

Parameters

- **n_peaks** (*int, optional*) – The number of peaks to identify, the default is 4.

- **as_Data** (*bool, optional*) – If *True* the peak info is returned as a *Data* object. The default is *False*.

Returns

peak_info (*numpy.ndarray*) – A two column numpy array with the with the location and the amplitude of each peak.

to_csv(*file_name*, *sep=','*)

This saves the data contained to a .csv file.

This saves the data in a csv file. It includes the interpolated, subtracted, inverse signals as well as the FFT. The FFT is interpolated to be the same length as the interpolated data.

Parameters

- **file_name** (*str*) – The name of the file to save the data to. If the file type is not one of '.csv', '.txt', or '.dat'; then '.csv' will be appended on to the end of the name.
- **sep** (*str, optional*) – The character used to the delineate between the data, the default is ','.

class gigaanalysis.qo.**QO_av**(*data_list*, *min_field*, *max_field*, *subtract_func*, *step_size=None*, *fft_cut=0*, *strip_nan=False*)

Bases: *QO*

Average Quantum Oscillation Class

This class applies a similar process to a set of sweeps in a list as the *QO* class. For each sweep the background in individually subtracted. They are then averaged to produce the FFT. The attributes for this class are also *Data* objects and these are the average of all the separately interpolated and subtracted sweeps.

This class is useful as the average of a collection of background subtractions are not necessarily the same as the subtract of their average.

One important point with this class is that the `raw` will be the given list as opposed to the average of this list. For the average it is best to use the `interp` attribute. The step_size if not given will also use the smallest of the generated step sizes form the list of raw sweeps.

Parameters

- **data** (*list*) – A list of raw data of the field sweep in the form of a list of *Data* objects to look for quantum oscillations in.
- **min_field** (*float*) – The lowest field value of the field range to inspect.
- **max_field** (*float*) – The highest field value of the field range to inspect.
- **subtract_func** (*calculable*) – This should take one *Data* object and return one *Data* of the same length. The input data is the interpolated sweep, and the output should be the data after the background has been subtracted.
- **step_size** (*float, optional*) – The size of field steps to interpolate. The default is 4 times the average step size in the raw data.
- **fft_cut** (*float, optional*) – The maximum frequency to consider in FFT, higher frequencies are dropped. THe default is to keep all the data.
- **strip_nan** (*bool, optional*) – If *True* non finite values are removed from the raw data. The default is *False* and this will raise an error is non finite values are in the raw data.

Variables

- **raw** (*list*) – The list of data originally given to the class.

- **interp** (Data) – The average of the sweeps cut to the field range and interpolated.
- **sub** (Data) – The average of the sweeps after the background subtraction.
- **invert** (Data) – The average of the subtracted sweeps after inverting the field values.
- **fft** (Data) – The Fourier transform of the inverted average signal.
- **min_field** (*float*) – The minimum field in the range in consideration.
- **max_field** (*float*) – The maximum field in the range in consideration.
- **step_size** (*float*) – The steps in field calculated in the interpolation.

make_QO(*raw_num*, *step_size=None*, *fft_cut=None*, *strip_nan=False*)

Make a Quantum Oscillation object form a certain sweep.

This return a new quantum oscillation object from a particular sweep in the raw list given. This will use the same field range and background subtraction as used in this class.

Parameters

- **raw_num** (*int*) – The number of the sweep to pass to *QO*. Like all python lists the counting starts at 0.
- **step_size** (*float, optional*) – If given this will be the step size used. If not given the step_size in the original class is used.
- **fft_cut** (*float, optional*) – If given this will be the FFT cut to used. If not given the fft_cut in the original class is used. To see the full range of frequencies set the fft_cut to 0.
- **strip_nan** (*bool, optional*) – This does nothing as in order to make this class there cannot be any NaNs left in the raw data. It is included for completeness.

Returns

single_QO (*QO*) – A quantum oscillation object with the same parameters as used in this class but only the data from one of the sweeps.

class gigaanalysis.qo.**QO_loess**(*data*, *min_field*, *max_field*, *loess_win*, *loess_poly*, *step_size=None*, *fft_cut=0*, *strip_nan=False*)

Bases: *QO*

Quantum Oscillation object with LOESS subtraction

This is a example of the *QO* which the subtraction using `mfunc.loess()`. The form is the same but with the initialising function takes the arguments to define the LOESS background subtraction.

Parameters

- **data** (Data) – The raw data of the field sweep to look for quantum oscillations in.
- **min_field** (*float*) – The lowest field value of the field range to inspect.
- **max_field** (*float*) – The highest field value of the field range to inspect.
- **loess_win** (*float*) – The length of the window in Tesla to use for the LOESS subtraction.
- **loess_poly** (*int*) – The order of the polynomial to use for the LOESS subtraction.
- **step_size** (*float, optional*) – The size of field steps to interpolate. The default is 4 times the average step size in the raw data.
- **fft_cut** (*float, optional*) – The maximum frequency to consider in FFT, higher frequencies are dropped. THe default is to keep all the data.

- **strip_nan** (*bool, optional*) – If *True* non finite values are removed from the raw data. The default is *False* and this will raise an error is non finite values are in the raw data.

:ivar

[This class has the same attributes as the *QO* class but also] with the information about the LOESS subtraction.

Variables

- **loess_win** (*float*) – The length of the window in Tesla to use for the LOESS subtraction.
- **loess_poly** (*int*) – The order of the polynomial to use for the LOESS subtraction.

class gigaanalysis.qo.**QO_loess_av**(*data_list*, *min_field*, *max_field*, *loess_win*, *loess_poly*, *step_size=None*, *fft_cut=0*, *strip_nan=False*)

Bases: *QO_av*

Average Quantum Oscillation object with LOESS subtraction

This is a example of the *QO_av* which the subtraction using `mfunc.loess()`. The form is the same but with the initialising function takes the arguments to define the LOESS background subtraction.

Parameters

- **data** (*list*) – A list of raw data of the field sweep in the form of a list of *Data* objects to look for quantum oscillations in.
- **min_field** (*float*) – The lowest field value of the field range to inspect.
- **max_field** (*float*) – The highest field value of the field range to inspect.
- **loess_win** (*float*) – The length of the window in Tesla to use for the LOESS subtraction.
- **loess_poly** (*int*) – The order of the polynomial to use for the LOESS subtraction.
- **step_size** (*float, optional*) – The size of field steps to interpolate. The default is 4 times the average step size in the raw data.
- **fft_cut** (*float, optional*) – The maximum frequency to consider in FFT, higher frequencies are dropped. THe default is to keep all the data.
- **strip_nan** (*bool, optional*) – If *True* non finite values are removed from the raw data. The default is *False* and this will raise an error is non finite values are in the raw data.

:ivar

[This class has the same attributes as the *QO* class but also] with the information about the LOESS subtraction.

Variables

- **loess_win** (*float*) – The length of the window in Tesla to use for the LOESS subtraction.
- **loess_poly** (*int*) – The order of the polynomial to use for the LOESS subtraction.

class gigaanalysis.qo.**QO_poly**(*data*, *min_field*, *max_field*, *poly_order*, *step_size=None*, *fft_cut=0*, *strip_nan=False*)

Bases: *QO*

Quantum Oscillation object with polynomial subtraction

This is a example of the *QO* which the subtraction using `mfunc.poly_reg()`. The form is the same but with the initialising function takes the arguments to define the polynomial background subtraction.

Parameters

- **data** (*Data*) – The raw data of the field sweep to look for quantum oscillations in.
- **min_field** (*float*) – The lowest field value of the field range to inspect.
- **max_field** (*float*) – The highest field value of the field range to inspect.
- **poly_order** (*int*) – The order of the polynomial to use for the subtraction.
- **step_size** (*float, optional*) – The size of field steps to interpolate. The default is 4 times the average step size in the raw data.
- **fft_cut** (*float, optional*) – The maximum frequency to consider in FFT, higher frequencies are dropped. THe default is to keep all the data.
- **strip_nan** (*bool, optional*) – If *True* non finite values are removed from the raw data. The default is *False* and this will raise an error is non finite values are in the raw data.

:ivar

[This class has the same attributes as the *QO* class but also] with the information about the polynomial subtraction.

Variables

poly_order (*int*) – The order of the polynomial to use for the subtraction.

class gigaanalysis.qo.**QO_poly_av**(*data_list*, *min_field*, *max_field*, *poly_order*, *step_size=None*, *fft_cut=0*, *strip_nan=False*)

Bases: *QO_av*

Average Quantum Oscillation object with polynomial subtraction

This is a example of the *QO_av* which the subtraction using `mfunc.poly_reg()`. The form is the same but with the initialising function takes the arguments to define the polynomial background subtraction.

Parameters

- **data** (*list*) – A list of raw data of the field sweep in the form of a list of *Data* objects to look for quantum oscillations in.
- **min_field** (*float*) – The lowest field value of the field range to inspect.
- **max_field** (*float*) – The highest field value of the field range to inspect.
- **poly_order** (*int*) – The order of the polynomial to use for the subtraction.
- **step_size** (*float, optional*) – The size of field steps to interpolate. The default is 4 times the average step size in the raw data.
- **fft_cut** (*float, optional*) – The maximum frequency to consider in FFT, higher frequencies are dropped. THe default is to keep all the data.
- **strip_nan** (*bool, optional*) – If *True* non finite values are removed from the raw data. The default is *False* and this will raise an error is non finite values are in the raw data.

:ivar

[This class has the same attributes as the *QO* class but also] with the information about the polynomial subtraction.

Variables

poly_order (*int*) – The order of the polynomial to use for the subtraction.

gigaanalysis.qo.**counting_freq**(*start_field*, *end_field*, *number_peaks*)

Counting quantum oscillation to obtain a frequency.

This returns the frequency of a quantum oscillation given a range of field and the number of osscilations that occur in that range. Performs `n*B1*B2/(B2 - B1)`

Parameters

- **start_field** (*float*) – The magnetic field to start the counting range.
- **end_field** (*float*) – The magnetic field to end the counting range.
- **number_peaks** (*float*) – The number of peaks in the given range.

Returns

frequency (*float*) – The frequency of the expected quantum oscillation in Tesla.

gigaanalysis.qo.**counting_num**(*start_field*, *end_field*, *frequency*)

Expected count of quantum oscillation at a given frequency.

This returns the number of quantum oscillations given a range of field and frequency of quantum oscillations in question. Performs `Freq*(B2 - B1)/(B1*B2)`

Parameters

- **start_field** (*float*) – The magnetic field to start the counting range.
- **end_field** (*float*) – The magnetic field to end the counting range.
- **frequency** (*float*) – The quantum oscillation frequency in Tesla.

Returns

number (*float*) – The number of expected peaks in this field range.

gigaanalysis.qo.**dingle_damping**(*fields*, *frequency*, *scatting*, *amp=1.0*, *as_Data=True*)

The Dingle Damping term from the LK formulas

This describes how the amplitude of quantum oscillations changes with applied field due to the scattering of electrons. The equation is of the form `r_d = amp*exp(-sqrt(2*pi*pi*hbar*frequency/qe)/(fields*scatting))`

Parameters

- **fields** (*numpy.ndarray*) – The values of magnetic field to be used when calculating the amplitude.
- **frequency** (*float*) – The frequency of the quantum oscillation in Tesla.
- **scatting** (*float*) – The scatting given by the mean free path in meters.
- **amp** (*float, optional*) – The amplitude at infinite field, the default is unity.
- **as_Data** (*bool, optional*) – If the default of *True* the result is returned as a *Data* object with the fields as the dependant variable. If *False* only the amplitudes are returned as a `numpy.ndarray`.

Returns

r_d (*Data, numpy.ndarray*) – The amplitude of the quantum oscillations as the field is changed.

gigaanalysis.qo.**lifshitz_kosevich**(*temps*, *field*, *e_mass*, *amp=1.0*, *as_Data=True*)

The Lifshitz-Kosevich formula

This formula describes the change the the amplitude of quantum oscillations as the temperature is changed. This is most commonly used to calculate the effective mass of a carrier if a temperature dependence of

quantum oscillations are taken. The equation is of the form `r_lk = amp*chi/sinh(chi)` where `chi = 2*pi*pi*kb*temps*me*e_mass/(hbar*qe*field)`.

Parameters

- **temps** (*numpy.ndarray*) – The value of temperatures to use to produce the LK curve.
- **field** (*float*) – The magnetic field value in Tesla of the applied field.
- **e_mass** (*float*) – The effective mass of the carriers in units of the electron mass.
- **amp** (*float, optional*) – The amplitude of the lowest temperature oscillations, the default is unity.
- **as_Data** (*bool, optional*) – If the default of *True* the result is returned as a *Data* object with the temps as the dependant variable. If false only the LK amplitudes are returned as a numpy.ndarray.

Returns

r_lk (*Data, numpy.ndarray*) – The amplitude of the quantum oscillations as the temperature is changed.

gigaanalysis.qo.**quantum_oscilation**(*fields*, *frequency*, *amp*, *phase*, *damping*, *as_Data=True*)

Example Quantum Oscillation

This is a simple example quantum oscillation for fitting and such like. I say simple because the amplitude and the damping term has no frequency or temperature dependence. This means you need to be more careful when thinking about the units but also makes fitting easier. The equation is of the form `quant_osc = amp*exp(-damping/fields)*sin(360*frequency/fields + phase)`

Parameters

- **fields** (*numpy.ndarray*) – The fields to produce the form quantum oscillation over.
- **frequency** (*float*) – The frequency of the quantum oscillation in Tesla.
- **amp** (*float*) – The amplitude of the quantum oscillation at infinite field.
- **phase** (*float*) – The phase of the quantum oscillation in degrees.
- **damping** (*float*) – The scatting damping of the quantum oscillation in Tesla.
- **as_Data** (*bool, optional*) – If the default of *True* the result is returned as a *Data* object with the fields as the dependant variable. If *False* only the amplitudes are returned as a numpy.ndarray.

Returns

quant_osc (*Data, numpy.ndarray*) – The amplitude of the quantum oscillations as the field is changed.

2.8 GigaAnalysis - Contour Mapping - gigaanalysis.contour

Here is a class and a few functions for contour mapping datasets to produce the gridded data for figures. This mostly makes use of a statistical technique called Gaussian Processes. In the simplest form this technique assumes that the surface being mapped is a combination of many normal distributions. While this is a crude approximation it works surprisingly well and sets a solid foundation for more complicated assumptions.

class gigaanalysis.contour.**GP_map**(*dataset*, *gen_x*, *gen_y*, *key_y=False*, *normalise_xy=False*, *look_up=None*, *even_space_y=None*)

Bases: object

Gaussian Process Contour Mapping

This takes a gigaanalysis dataset (a `dict` of *`Data`* objects), and two `numpy.ndarray` of x and y values to interpolate over in a grid. It then uses the method of Gaussian Processes to interpolate from the provided data into the generated values on the gird.

The class requires the kernel for the Gaussian process to be set using one of two methods. For most applications it is sufficient to use the method *`GP_map.set_distance_kernel()`* as this only needs one argument. This does assume that the kernel can be expressed as the euclidean distance between the point of interest and the known data. The method *`GP_map.set_xy_kernel()`* can be used for more complicated kernel application.

Finally `GP_map.predict_z()` can be run to generate the interpolated points. This is separated into its own method as it contains the computationally heavy part of the calculation. If desired the generated data can be cut to the region contained in a convex hull of the provided data to avoid unintentional extrapolation.

Parameters

- **dataset** (*dict of {float:Data} or numpy.ndarray*) – The data to perform the interpolation on. This can be in the form of a gigaanalysis dataset where the keys of the dictionaries are floats (unless 'look_up' is used). This will then be unrolled using `parse.unroll_dataset()`. A three column numpy array can also be provided with the x, y, and z values in each respective column.
- **gen_x** (*numpy.ndarray*) – A 1D numpy array with the x values to interpolate.
- **gen_y** (*numpy.ndarray*) – A 1D numpy array with the y values to interpolate.
- **key_y** (*bool, optional*) – If default of *False* then the keys of the array are used as the x values, and the x values of the *`Data`* objects are used as the y values. To swap these two roles set key_y to *True*.
- **normalise_xy** (*bool or tuple, optional*) – If 'True' then the x y values are normalised to the range 0 to 1. This is useful for the kernel to deal with the probable disparity of units in the two directions. If default of 'False' then this is not done. A tuple of two floats can be provided instead which the x and y values will be normalised to instead of range unity. This can be useful for weighting the units in an euclidean kernel.
- **look_up** (*dict, optional*) – This is a dictionary with all the keys that match the keys in the dataset and values which are floats to be used for the x values. This is passed to `parse.unroll_dataset()`. Default is *None* and then keys are used as the values.
- **even_space_y** (*float, optional*) – If a float is provided then the independent data in the gigaanalysis data objects is evenly spaced using `Data.interp_step()`. This is useful if more finely spaced data points shouldn't be given more weight in the calculation. The default is none and then the original data is used.

Variables

- **input_x** (*numpy.ndarray*) – A 1D array of x values of the provided data to process. These will be normalised if *normalise_xy* is *True*.
- **input_y** (*numpy.ndarray*) – A 1D array of y values of the provided data to process. These will be normalised if *normalise_xy* is *True*.
- **input_z** (*numpy.ndarray*) – A 1D array of z values of the provided data to process.
- **gen_x** (*numpy.ndarray*) – A 1D array of the x values to interpolate.
- **gen_y** (*numpy.ndarray*) – A 1D array of the y values to interpolate.
- **gen_xx** (*numpy.ndarray*) – A 2D array of the x values for all the interpolated points. These will be normalised if *normalise_xy* is *True*.

- **gen_yy** (*numpy.ndarray*) – A 2D array of the y values for all the interpolated points. These will be normalised if *normalise_xy* is *True*.
- **kernel** (*Callable*) – A function which takes four 1D numpy arrays. The first is a set of x values and then corresponding y values and then another similar pair. These are then used to produce a 2D array of the kernel weights.
- **kernel_args** (*dict*) – This is a dictionary of keyword argument to be passed to the provided kernel function.
- **white_noise** (*float*) – The amplitude of the white noise term in the kernel function.
- **kmat_inv** (*numpy.ndarray*) – A 2D array which is the independent part of the covariance matrix.
- **predict_z** (*numpy.ndarray*) – A 2D array with the interpolated values produced.

calculate_log_mlh()

Calculate and return the negative log marginal likelihood.

This is the scaler that needs to be minimised to compare the values of kernel parameters. This recalculates the values in a way to try and speed things in the minimisation process.

Returns

neg_log_marg_lh (*float*) – The negative log marginal likelihood for the current kernel and data provided.

cap_min_max(*z_min*, *z_max*)

Caps the z values between a minimum and maximum values

This changed the `predict_z` attribute so that values above and bellow a range are caped to the values. This can be useful for trimming unphysical values or cutting out extremes from extrapolation.

Parameters

- **z_min** (*float, None*) – If a float is given then the all the values bellow this value are changed to equal this value. If *None* is given then a cap isn't preformed.
- **z_max** (*float, None*) – If a float is given then the all the values above this value are changed to equal this value. If *None* is given then a cap isn't preformed.

cut_outside_hull(*tol=1e-09*)

Removes data that requires extrapolation

When called this function sets all the values to interpolate that are not surrounded by three points from the input data to *numopy.nan*. This means that the result doesn't extrapolate. This is done using `scipy.spatial.ConvexHull`.

This changes the value of `predict_z`.

Parameters

tol (*float, optional*) – The tolerance when comparing points to see if they are inside the convex hull. A higher tolerance means more points will be included. The default is `10**(-9)`.

optermise_argument(*arguments*, ***kwargs*)

Minimise the negative log marginal likelihood.

This uses `scipy.optimize.minimize()` to change the value of keyword arguments to minimise the value from *calculate_log_mlh()*. This should take both into account the model complexity and the quality of the kit to the data. Keyword arguments are passed to `scipy.optimize.minimize()`, a very useful one is *bounds* which is a list of tuples of the lower then upper bounds.

Parameters

arguments (*dict*) – A dictionary of the keywords for the kernel function and the initial values to start the optimisation from.

Returns

minimize_result (*scipy.optimize.OptimizeResult*) – The result from the running of `scipy.optimize.minimize()`, the *x* argument of the result is also set to the `kernel_args` attribute.

plot_contour(*colorbar_kwargs={}*, ***kwargs*)

Plot the generated data as a contour map

This makes use of `matplotlib.pyplot.contor()` and keyword arguments are passed to it. Keyword arguments can be passed to the colour bar by setting the keyword argument *colorbar_kwargs* to a dictionary. This uses `matplotlib.pyplot.colorbar()`.

plot_contourf(*colorbar_kwargs={}*, ***kwargs*)

Plot the generated data as a contour map

This makes use of `matplotlib.pyplot.contorf()` and keyword arguments are passed to it. Keyword arguments can be passed to the colour bar by setting the keyword argument *colorbar_kwargs* to a dictionary. This uses `matplotlib.pyplot.colorbar()`.

plot_input_scatter(***kwargs*)

Plots the input data as a scatter graph

This is useful for debugging and getting an idea of what the data is like before applying Gaussian processes. Makes use of `matplotlib.pyplot.scatter()`, and keyword arguments are passed to it.

plotting_arrays()

Produces the three arrays need for plotting

This makes use of `numpy.meshgrid(gen_x, gen_y)`, and is in the from needed for `matplotlib.pyplot.contorf()`.

Returns

- **r_gen_xx** (*numpy.ndarray*) – A 2D array of the x values. These are not normalised even if a normalisation is set.
- **r_gen_yy** (*numpy.ndarray*) – A 2D array of the y values. These are not normalised even if a normalisation is set.
- **predict_z** (*numpy.ndarray*) – A 2D array of the z values. This is the same as `predict_z`.

predict(*cut_outside=False*, *new_invert=False*, *no_return=False*, *cap_z=None*)

Calculates the interpolated z values.

Runs the calculation and returns the result of interpolating the z values using the Gaussian processes technique.

Parameters

- **cut_outside** (*bool, optional*) – If default of 'False' returns all the data for the grid to interpolate. If 'True' the values that require extrapolation are set to numpy.nan. This is done using *cut_outside_hull()*. If float is given then that is used as the tolerance and the cut is preformed.
- **new_invert** (*bool, optional*) – If 'True' then the kernel will be inverted again. If the default of 'False' then the kernel will only be recalculated if it has been updated. If the kernel is updated by addressing the attribute then to be recalculated this need to be set to 'True' for the new kernel to be used.

- **no_return** (*bool, optional*) – If the default of 'False' the prediction will be returned. If 'True' then nothing will be.
- **cap_z** – If a tuple of floats is given then the `predict_z` is capped between the two values given using *cap_min_max()*. If None is given then the cap isn't performed.
- **tuple** – If a tuple of floats is given then the `predict_z` is capped between the two values given using *cap_min_max()*. If None is given then the cap isn't performed.

Returns

predict_z (*numpy.ndarray*) – A 2D array of the values of the calculated z values in the locations of `gen_x` and `gen_y`. This function also sets the attribute `predict_z`.

set_distance_kernel(*dis_kernel*, *white_noise*, ***kernel_args*)

Set a kernel which is a function of euclidean distance.

Here you can set a kernel which is a function of distance between the point to interpolate and the known data. The kernel is a calculable function with one argument. The keyword arguments are passed to the distance kernel function.

Parameters

- **dis_kernel** (*Callable*) – A function with one argument which takes a `numpy.ndarray` of distance values and returns the same shaped `numpy.ndarray` of kernel weights. Keyword arguments will also be passed to this function.
- **white_noise** (*float*) – The value of the white noise term which takes into account stochastic error in the sample. It also insures the success of the matrix inversion, so even with perfect data a small white noise term is preferable.

set_kernel_args(***kernel_args*)

Set keyword arguments for the kernel function.

This allows new kernel keyword values to be set without resupplying the kernel and all the other keyword values. This keeps the values already set unless they are written over. The values are supplied by providing keyword arguments to this function.

set_xy_kernel(*xy_kernel*, *white_noise*, ***kernel_args*)

Set a kernel which is a function of the x and y values.

Here you can set a kernel which is a function of the x and y value of both terms to compare. The kernel is a calculable function with four arguments. The keyword arguments are passed to the distance kernel function.

Parameters

- **dis_kernel** (*Callable*) – A function with four arguments which takes four `numpy.ndarray` of x and y values of the same shape and returns the same shaped `numpy.ndarray` of kernel weights. The arguments are x1, y1, x2, y2 where x1 and y1 are the values of the first coordinates to compare, and x2 and y1 are the second. Keyword arguments will also be passed to this function.
- **white_noise** (*float*) – The value of the white noise term which takes into account stochastic error in the sample. It also insures the success of the matrix inversion, so even with perfect data a small white noise term is preferable.

gigaanalysis.contour.**elliptical_gaussian_kernel**(*x1*, *y1*, *x2*, *y2*, *const=0.0*, *amp=1.0*, *x_length=1.0*, *y_length=1.0*, *angle=0.0*)

An elliptical Gaussian kernel for contour fitting.

This is a simple Gaussian kernel for use with *GP_map.set_xy_kernel()*. The keyword arguments can be set when they are passed through *GP_map.set_xy_kernel()*.

Parameters

- **x1** (*numpy.ndarray*) – The four arguments are arrays which contain the x and y values from the points to generate the appropriate kernel matrix. These arrays are all the same size.
- **y1** (*numpy.ndarray*) – See above.
- **x2** (*numpy.ndarray*) – See above.
- **y2** (*numpy.ndarray*) – See above.
- **const** (*float, optional*) – A constant term that changes the background level. Default is `0`
- **amp** (*float, optional*) – The amplitude of the Gaussian. The default is `1`
- **x_length** (*float, optional*) – The length scale of the x component Gaussian. The default is `1`
- **y_length** (*float, optional*) – The length scale of the y component Gaussian. The default is `1`
- **angle** (*float, optional*) – The angle in radians to rotate the x and y contributions the default is `0` which keeps the x and y values independent.

Returns

kernel_mat (*numpy.ndarray*) – A `numpy.ndarray` the same size as the input arrays with the kernel matrix elements.

`gigaanalysis.contour.`**gaussian_kernel**(*x1*, *y1*, *x2*, *y2*, *const=0.0*, *amp=1.0*, *length=1.0*)

A Gaussian kernel for contour fitting.

This is a simple Gaussian kernel for use with *GP_map.set_xy_kernel()*. It has an equation of the form `K = const + amp*exp(((x1 - x2)**2 + (y1 - y2)**2)/length**2)`. The keyword arguments can be set when they are passed through *GP_map.set_xy_kernel()*.

Parameters

- **x1** (*numpy.ndarray*) – The four arguments are arrays which contain the x and y values from the points to generate the appropriate kernel matrix. These arrays are all the same size.
- **y1** (*numpy.ndarray*) – See above.
- **x2** (*numpy.ndarray*) – See above.
- **y2** (*numpy.ndarray*) – See above.
- **const** (*float, optional*) – A constant term that changes the background level. Default is `0`
- **amp** (*float, optional*) – The amplitude of the Gaussian. The default is `1`
- **length** (*float, optional*) – The length scale of the Gaussian. The default is `1`

Returns

kernel_mat (*numpy.ndarray*) – A `numpy.ndarray` the same size as the input arrays with the kernel matrix elements.

`gigaanalysis.contour.`**linear_kernel**(*x1*, *y1*, *x2*, *y2*, *const=1.0*, *amp=1.0*, *x_scale=1.0*, *y_scale=1.0*)

A linear kernel for contour fitting.

This is a simple linear kernel for use with *GP_map.set_xy_kernel()*. It has an equation of the form `K = const + amp*(x1*x2/x_scale**2+ y1*y2/y_scale**2)`. The keyword arguments can be set when they are passed through *GP_map.set_xy_kernel()*. There are much faster ways of doing this than with Gaussian

processes the utility of this function is to be combined with others. Also amp, x_scale, and y_scale over define the function so don't optimise on all at once; they are included to help to relate to physical properties.

Parameters

- **x1** (*numpy.ndarray*) – The four arguments are arrays which contain the x and y values from the points to generate the appropriate kernel matrix. These arrays are all the same size.
- **y1** (*numpy.ndarray*) – See above.
- **x2** (*numpy.ndarray*) – See above.
- **y2** (*numpy.ndarray*) – See above.
- **const** (*float, optional*) – A constant term that changes the background level. The default is `0`
- **amp** (*float, optional*) – The amplitude of the linear term. The default is 1
- **x_scale** (*float, optional*) – The scaling of the x values in the same units as x. The default is 1.
- **y_scale** (*float, optional*) – The scaling of the y values in the same units as y. The default is 1.

Returns

kernel_mat (*numpy.ndarray*) – A `numpy.ndarray` the same size as the input arrays with the kernel matrix elements.

gigaanalysis.contour.**rational_quadratic_kernel**(*x1*, *y1*, *x2*, *y2*, *const=0.0*, *amp=1.0*, *length=1.0*, *scale=1.0*)

A rational quadratic kernel for contour fitting.

This is a rational quadratic kernel for use with *GP_map.set_xy_kernel()*. It has an equation of the form It has an equation of the form `K = const + amp*(1 + ((x1 - x2)**2 + (y1 - y2)**2)/2/length**2/scale)**scale`. The keyword arguments can be set when they are passed through *GP_map.set_xy_kernel()*. This can be thought of a combination of many different Gaussian kernels to different powers. These are the same when the scale goes to infinity.

Parameters

- **x1** (*numpy.ndarray*) – The four arguments are arrays which contain the x and y values from the points to generate the appropriate kernel matrix. These arrays are all the same size.
- **y1** (*numpy.ndarray*) – See above.
- **x2** (*numpy.ndarray*) – See above.
- **y2** (*numpy.ndarray*) – See above.
- **const** (*float, optional*) – A constant term that changes the background level. The default is `0`
- **amp** (*float, optional*) – The amplitude of the rational quadratic term. The default is 1
- **length** (*float, optional*) – The length scale of the kernel. The default is 1
- **scale** (*float, optional*) – The scaling function between order terms. The default is ''1"

Returns

kernel_mat (*numpy.ndarray*) – A `numpy.ndarray` the same size as the input arrays with the kernel matrix elements.

2.9 GigaAnalysis - Superconductors - gigaanalysis.htsc

Here are a few functions for equations that are useful for high temperature superconducting science. These are useful for getting doping values from transition temperatures and vice-versa. The default values for these are given for YBCO using the values from DOI: 10.1103/PhysRevB.73.180505 Also extracting the transition temperature from stepped data.

gigaanalysis.htsc.**cdw_factor**(*doping*, *a_cdw=- 0.204*, *p_cdw=11.874*, *w_cdw=3.746*, *as_Data=False*)

Calculates the faction the critical temperature is reduced by CDW.

This is for calculating the ratio of critical temperature suppression from the Charge Density Wave. The values are for YBCO.

Parameters

- **doping** (*float or numpy.ndarray*) – The value or values of the doping to calculate the critical temperature suppression for. The units are percent of holes per unit cell per plane.
- **a_cdw** (*float : optional*) – The maximum critical temperature suppression of the dome as a ratio. The value is absolute so negative values are a suppression and the default is -0.204.
- **p_cdw** (*float, optional*) – The doping at the maximum amount of CDW in percent. The default value is 11.874 %.
- **w_cdw** (*float, optional*) – The full width half maximum of the CDW dome in percent doping. The default value is 3.764 %.
- **as_Data** (*bool, optional*) – If 'True' a *Data* object is returned with the dopings as the dependent variable and the critical temperatures suppression as the independent variable. The default is 'False' which returns a numpy.ndarray.

Returns

critical_temperature (*float, numpy.ndarray, or Data*) – The values of the critical temperature suppression of the superconductor due to the CDW at the given doping values as a ratio.

gigaanalysis.htsc.**dome_p2tc**(*doping*, *t_max=94.3*, *p_max=16.0*, *p_w=11.0*, *as_Data=False*)

This converts values of doping to transition temperature on a SC dome.

The default parameters from the dome are taken from YBCO, but are changeable.

Parameters

- **doping** (*float or numpy.ndarray*) – The value or values of the doping to calculate the critical temperature for. The units are percent of holes per unit cell per plane.
- **t_max** (*float : optional*) – The maximum critical temperature of the dome in Kelvin. The default is 94.3 K.
- **p_max** (*float, optional*) – The doping at the maximum critical temperature in percent. The default is 16 %.
- **p_w** (*float, optional*) – The half width of the dome in percent doping. The default value is 11 %.
- **as_Data** (*bool, optional*) – If 'True' a *Data* object is returned with the dopings as the dependent variable and the critical temperatures as the independent variable. The default is 'False' which returns a numpy.ndarray.

Returns

critical_temperature (*float, numpy.ndarray, or Data*) – The values of the critical temperature of the superconductor at the given doping values in Kelvin.

gigaanalysis.htsc.**dome_tc2p**(*critical_temperature*, *side*, *t_max=94.3*, *p_max=16.0*, *p_w=11.0*, *as_Data=False*)

This converts values of the critical temperature to doping.

The default parameters from the dome are taken from YBCO, but are changeable.

Parameters

- **critical_temperature** (*float or numpy.ndarray*) – The value or values of the critical temperature to calculate the doping for. The units are percent of holes per unit cell per plane.
- **side** (*str or numpy.ndarray of {'UD' or 'OD'}*) – The side of the dome to calculate the doping of. 'UD' for the under doped size, and 'OD' for the over doped side. This is either a string or an array the length of the given critical temperatures.
- **t_max** (*float : optional*) – The maximum critical temperature of the dome in Kelvin. The default is 94.3 K.
- **p_max** (*float, optional*) – The doping at the maximum critical temperature in percent. The default is 16 %.
- **p_w** (*float, optional*) – The half width of the dome in percent doping. The default value is 11 %.
- **as_Data** (*bool, optional*) – If 'True' a *Data* object is returned with the critical temperatures as the dependent variable and the dopings as the independent variable. The default is 'False' which returns a numpy.ndarray.

Returns

doping (*float, numpy.ndarray, or Data*) – The values of the doping of the superconductor with the given critical temperature and side of the dome.

gigaanalysis.htsc.**trans_res**(*data*, *res_onset*, *under_nan=False*, *over_nan=False*, *as_ratio=False*)

Returns the dependent variable value at the resistive transition.

This assumes the data is sorted and returns the last value that is below the onset resistance specified.

Parameters

- **data** (Data) – The sorted resistivity data to look for the transition in.
- **res_onset** (*float*) – The value of resistivity that if measured is then considered that the sample is now not superconducting. This is in the same units as given in the data.
- **under_nan** (*bool, optional*) – If the default of 'False' zero is returned if the all the data lays above the onset value. If 'True' NaN is returned.
- **over_nan** (*bool, optional*) – If the default of 'False' the last value is returned if all the data lays below the onset value. If 'True' NaN is returned.
- **as_ratio** (*bool, optional*) – If *True* then the value of res_onset is multiplied by the maximum value of the data. For most examples this means that if `res_onset = 0.01` then the transition would be at 1% of the high temperature value. The default is *False*.

Returns

transition_onset (*float*) – The last value of the dependent variable where the independent variable is below the given onset value.

gigaanalysis.htsc.**ybco_p2tc**(*doping*, *t_max=94.3*, *p_max=16.0*, *p_w=11.0*, *a_cdw=- 0.204*, *p_cdw=11.874*, *w_cdw=3.746*, *as_Data=False*)

This converts values of doping to transition temperature of YBCO.

This calculates the transition temperature from the doping while taking into consideration the effect of CDW. The default parameters from the dome are taken from YBCO, but are changeable.

Parameters

- **doping** (*float or numpy.ndarray*) – The value or values of the doping to calculate the critical temperature for. The units are percent of holes per unit cell per plane.
- **t_max** (*float : optional*) – The maximum critical temperature of the dome in Kelvin. The default is 94.3 K.
- **p_max** (*float, optional*) – The doping at the maximum critical temperature in percent. The default is 16 %.
- **p_w** (*float, optional*) – The half width of the dome in percent doping. The default value is 11 %.
- **a_cdw** (*float : optional*) – The maximum critical temperature suppression of the dome as a ratio. The value is absolute so negative values are a suppression and the default is -0.204.
- **p_cdw** (*float, optional*) – The doping at the maximum amount of CDW in percent. The default value is 11.874 %.
- **w_cdw** (*float, optional*) – The full width half maximum of the CDW dome in percent doping. The default value is 3.764 %.
- **as_Data** (*bool, optional*) – If 'True' a *Data* object is returned with the dopings as the dependent variable and the critical temperatures as the independent variable. The default is 'False' which returns a numpy.ndarray.

Returns

critical_temperature (*float, numpy.ndarray, or Data*) – The values of the critical temperature of the superconductor at the given doping values in Kelvin.

gigaanalysis.htsc.**ybco_tc2p**(*critical_temperature*, *side*, *t_max=94.3*, *p_max=16.0*, *p_w=11.0*, *a_cdw=-0.204*, *p_cdw=11.874*, *w_cdw=3.746*, *gen_points=500*, *as_Data=False*)

This converts values of the critical temperature to doping.

This takes into consideration the Charge Density Wave (CDW) found in YBCO. The default parameters from the dome are taken from YBCO, but are changeable.

Parameters

- **critical_temperature** (*float or numpy.ndarray*) – The value or values of the critical temperature to calculate the doping for. The units are percent of holes per unit cell per plane.
- **side** (*str or numpy.ndarray of {'UD' or 'OD'}*) – The side of the dome to calculate the doping of. 'UD' for the under doped size, and 'OD' for the over doped side. This is either a string or an array the length of the given critical temperatures.
- **t_max** (*float : optional*) – The maximum critical temperature of the dome in Kelvin. The default is 94.3 K.
- **p_max** (*float, optional*) – The doping at the maximum critical temperature in percent. The default is 16 %.
- **p_w** (*float, optional*) – The half width of the dome in percent doping. The default value is 11 %.

- **a_cdw** (*float : optional*) – The maximum critical temperature suppression of the dome as a ratio. The value is absolute so negative values are a suppression and the default is -0.204.
- **p_cdw** (*float, optional*) – The doping at the maximum amount of CDW in percent. The default value is 11.874 %.
- **w_cdw** (*float, optional*) – The full width half maximum of the CDW dome in percent doping. The default value is 3.764 %.
- **gen_points** (*int, optional*) – The doping values are calculated by interpolating along a curve. This parameters specifies how many points to generate for the interpolation. THe default is 500.
- **as_Data** (*bool, optional*) – If 'True' a *Data* object is returned with the critical temperatures as the dependent variable and the dopings as the independent variable. The default is 'False' which returns a numpy.ndarray.

Returns

doping (*float, numpy.ndarray, or Data*) – The values of the doping of the superconductor with the given critical temperature and side of the dome.

2.10 GigaAnalysis - Magnetism - gigaanalysis.magnetism

Here are a few functions for equations that are useful for magnetism science. They can be made to produce a Data object or just a numpy.ndarray. This works well with the fitting module.

gigaanalysis.magnetism.**brillouin_function**(*fields*, *n_ion*, *g*, *j*, *temp*, *as_Data=False*)

The Brillouin function

This function which describes the magnetisation of an ideal paramagnet composed of ions with a certain spin J.

Parameters

- **fields** (*float or numpy.ndarray*) – The value or values of the applied magnetic filed in Tesla.
- **n_ion** (*float*) – The number of contributing ions to the magnetism.
- **g** (*float*) – The ions g factor or dimensionless magnetic moment.
- **i** (*float*) – Is a positive integer or half integer which is the spin of the ions. This function does not but constrains on the value of j.
- **temp** (*float*) – Temperature in Kelvin.
- **as_Data** (*bool, optional*) – If False returns a numpy.ndarray which is the default behaviour. If True returns a *gigaanalysis.data.Data* object with the fields values given and the cosponsoring magnetisation.

Returns

Magnetisation (*numpy.ndarray, Data*) – The magnetisation produced in units of J/T.

gigaanalysis.magnetism.**langevin_function**(*fields*, *n_ion*, *g*, *temp*, *as_Data=False*)

The Langevin function

This is the classical limit of the Brillouin function which describes the magnetisation of an ideal paramagnet.

Parameters

- **fields** (*float or numpy.ndarray*) – The value or values of the applied magnetic filed in Tesla
- **n_ion** (*float*) – The number of contributing ions to the magnetism
- **g** (*float*) – The ions g factor or dimensionless magnetic moment
- **temp** (*float*) – Temperature in Kelvin
- **as_Data** (*bool, optional*) – If False returns a numpy.ndarray which is the default behaviour. If True returns a *gigaanalysis.data.Data* object with the fields values given and the cosponsoring magnetisation.

Returns

Magnetisation (*numpy.ndarray, Data*) – The magnetisation produced in units of J/T

2.11 GigaAnalysis - Heat Capacity - gigaanalysis.heatc

Here are a few functions for equations that are useful for heat capacity measurements. They can be made to produce a Data object or just a numpy.ndarray. This works well with the fitting module.

gigaanalysis.heatc.**schottky_anomaly**(*temps*, *num*, *gap*, *as_Data=False*)

The Schottky anomaly

The function which describes the heat capacity of a two state system.

Parameters

- **temps** (*float or numpy.ndarray*) – The value or values of the temperature in Kelvin.
- **num** (*float*) – The number of moles of states contributing.
- **gap** (*float*) – The energy gap between the two states in Joules.
- **as_Data** (*bool, optional*) – If False returns a numpy.ndarray which is the default behaviour. If True returns a *gigaanalysis.data.Data* object with the fields values given and the cosponsoring magnetisation.

Returns

Heat Capacity (*numpy.ndarray, Data*) – The heat capacity in units of J/K/mol.

2.12 GigaAnalysis - Digital Lock In - gigaanalysis.diglock

This program is to recreate what a lock in would do for slower measurements but for our high field experiments. This is based around what the program in DRS and WUH did. This module also includes the *scanning_fft()* which is used for PDO and TDO measurements.

gigaanalysis.diglock.**butter_bandpass**(*lowcut*, *highcut*, *fs*, *order=5*)

Produces the polynomial values for the Butterworth bandpass.

This make use of scipy.signal.butter(), and supplies values for scipy.signal.filtfilt().

Parameters

- **lowcut** (*float*) – The low frequency cut off.
- **highcut** (*float*) – The high frequency cut off.
- **fs** (*float*) – The sample frequency of the data.

- **order** (*int, optional*) – The order of the Butterworth filter, default is 5.

Returns

- *b, numpy.ndarray,* – The numerator of the polynomials of the IIR filter.
- *a, numpy.ndarray,* – The denominator of the polynomials of the IIR filter.

gigaanalysis.diglock.**butter_bandpass_filter**(*data*, *lowcut*, *highcut*, *fs*, *order=5*)

Applies a Butterworth bandpass filter to a set of data.

This makes use of *butter_bandpass()* and applied that filter to a given signal.

Parameters

- **data** (*numpy.ndarray*) – A array containing the signal.
- **lowcut** (*float*) – The low frequency cut off.
- **highcut** (*float*) – The high frequency cut off.
- **fs** (*float*) – The sample frequency of the data points.
- **order** (*int, optional*) – The order of the filter to apply, default is 5.

Returns

filtered (*numpy.ndarray*) – The filtered data.

gigaanalysis.diglock.**find_freq**(*data*, *samp_freq*, *padding=1*, *fit_point=3*, *plot=False*, *amp=False*, *skip_start=40*)

Finds the dominate frequency in oscillatory data.

It performs an FFT and then finds the maximal frequency using *polypeak()*.

Parameters

- **data** (*numpy.ndarray*) – The signal in evenly spaced points
- **samp_freq** (*float*) – The measurement frequency of the data points
- **padding** (*float, optional*) – Pads the data my multiplying it before the FFT, default is 1.
- **fit_point** (*int, optional*) – Number of fit points to be used in *polypeak()*, default is 3.
- **plot** (*bool, optional*) – If *True* plots a figure to check the identification of the peak.
- **amp** (*bool, optional*) – If *True* returns the FFT amplitude of the frequency as well.
- **skip_start** (*int, optional*) – The number of points to skip the low frequency tail of the FFT, the default is 40.

Returns

- **peak_freq** (*float*) – The value of the dominate frequency.
- **peak_amp** (*float*) – If amp is *True* also returns the amplitude of the dominate frequency.

gigaanalysis.diglock.**find_phase**(*data*, *fs*, *freq*)

Finds the phase of a oscillatory signal.

Parameters

- **data** (*numpy.ndarray*) – An array containing the signal.
- **fs** (*float*) – Sample frequency of the data points.

- **freq** (*float*) – The frequency of the oscillatory signal.

Returns

phase (*float*) – The phase in degrees of the oscillatory signal.

gigaanalysis.diglock.**flat_lock_in**(*signal*, *time_const*, *fs*, *freq*, *phase*)

Performs a lock in of the signal and averages with a flat window.

The window from *flat_window()* is convolved with the signal that has been multiplied by the reference from *gen_ref()*.

Parameters

- **signal** (*numpy.ndarray*) – The oscillatory signal to lock in to.
- **time_const** (*float*) – The time constant for the averaging.
- **fs** (*float*) – The sample frequency of the signal data.
- **freq** (*float*) – The frequency of the oscillatory signal to lock in to.
- **phase** (*float*) – The phase of the signal is degrees.

Returns

signal_amp (*numpy.ndarray*) – The signal after the lock in processes which is equal to the amplitude of the oscillatory signal at the given frequency.

gigaanalysis.diglock.**flat_window**(*time_const*, *fs*, *freq*)

Produces a flat window for averaging.

Uses *round_oscillation()* to set the window as the same length as a whole number of oscillations.

Parameters

- **time_const** (*float*) – The time for averaging
- **fs** (*float*) – The sample frequency of the signal
- **freq** (*float*) – The frequency of the oscillatory signal.

Returns

flat_window (*numpy.ndarray*) – An array to convolve with the signal that has a unit total.

gigaanalysis.diglock.**gen_ref**(*freq*, *fs*, *phase*, *number*)

Produces the reference signal for the digital lock in.

Parameters

- **freq** (*float*) – Frequency of the signal.
- **fs** (*float*) – Sample frequency of the data.
- **phase** (*float*) – Phase of the signal in degrees.
- **number** (*int*) – The number of points to generate.

Returns

ref_signal (*numpy.ndarray*) – An array containing the reference signal.

gigaanalysis.diglock.**ham_lock_in**(*signal*, *time_const*, *fs*, *freq*, *phase*)

Performs a lock in of the signal and averages with a hamming window.

The window from *hamming_window()* is convolved with the signal that has been multiplied by the reference from *gen_ref()*.

Parameters

- **signal** (*numpy.ndarray*) – The oscillatory signal to lock in to.
- **time_const** (*float*) – The time constant for the averaging.
- **fs** (*float*) – The sample frequency of the signal data.
- **freq** (*float*) – The frequency of the oscillatory signal to lock in to.
- **phase** (*float*) – The phase of the signal is degrees.

Returns

signal_amp (*numpy.ndarray*) – The signal after the lock in processes which is equal to the amplitude of the oscillatory signal at the given frequency.

gigaanalysis.diglock.**hamming_window**(*time_const*, *fs*)

Produces a hamming window for averaging the signal.

Uses a Hamming filter shape `scipy.signal.hamming()`.

Parameters

- **time_const** (*float*) – The time for averaging, this is like the 'mean' time.
- **fs** (*float*) – The sample frequency of the data.

Returns

hamming_window (*numpy.ndarray*) – An array to convolve with the signal that has unit total.

gigaanalysis.diglock.**phase_in**(*signal_in*, *signal_out*, *aim='change'*, ***kwargs*)

Picks a phase that maximises something about the signal.

This makes use of either *phase_in_change()* or `phase_in_value', depending on the aim keyword. Uses :func:`scipy.optimize.minimize()` and keyword arguments are passed to it.

Parameters

- **signal_in** (*numpy.ndarray*) – The values containing the in phase signal.
- **signal_out** (*numpy.ndarray*) – The values containing the out of phase signal needs to be the same shape as *signal_in*.
- **aim** (*str, {'change', 'value'}, optional*) – What to maximise. The default is 'change'.

Returns

max_phase (*float*) – The best phase in degrees between 0 deg and 360 deg.

gigaanalysis.diglock.**phase_in_change**(*signal_in*, *signal_out*, ***kwargs*)

Picks a phase to capture the majority of the change of the signal.

This given an in and out of phase signal returns the phase shift to move the majority of the change in signal into the in phase component. It also chooses the phase shift so the signal is positive and between 0 and 360 deg. Uses `scipy.optimize.minimize()` and keyword arguments are passed to it.

Parameters

- **signal_in** (*numpy.ndarray*) – The values containing the in phase signal.
- **signal_out** (*numpy.ndarray*) – The values containing the out of phase signal needs to be the same shape as *signal_in*.

Returns

max_phase (*float*) – The phase in degrees between 0 deg and 360 deg where the change in signal in the out of phase is minimised.

gigaanalysis.diglock.**phase_in_value**(*signal_in*, *signal_out*, ***kwargs*)

Picks a phase to capture the majority of the amplitude of the signal.

This given an in and out of phase signal returns the phase shift to move the majority of the signal into the in phase component. It also chooses the phase shift so the signal is positive and between 0 and 360 deg. Uses `scipy.optimize.minimize()` and keyword arguments are passed to it.

Parameters

- **signal_in** (*numpy.ndarray*) – The values containing the in phase signal.
- **signal_out** (*numpy.ndarray*) – The values containing the out of phase signal needs to be the same shape as *signal_in*.

Returns

max_phase (*float*) – The phase in degrees between 0 deg and 360 deg where amplitude of the signal is maximised.

gigaanalysis.diglock.**polypeak**(*signal*, *fit_point=3*, *low_f_skip=0*)

Finds the largest value in a data set by fitting a parabola.

It picks the largest point in the dataset and fits a quadratic parabola to it. It then uses that to get the interpolated maximum.

Parameters

- **signal** (*numpy.ndarray*) – The data to interpolate the highest value of.
- **fit_point** (*int, optional*) – The number of points to use in the fit. Needs to be odd.
- **low_f_skip** (*int, optional*) – The number of points to disregard at the start of the data.

Returns

- **x** (*float*) – The x position as a rational number in relation to the index of the maximal value.
- **y** (*float*) – The y position of the interpolated maximum value.

gigaanalysis.diglock.**round_oscillation**(*time_const*, *freq*)

Rounds to nearest number of whole oscillations.

Used for minimising aliasing issues.

Parameters

- **time_const** (*float*) – The averaging time
- **freq** (*float*) – Frequency of the signal

Returns

number_osc (*int*) – The closet number of oscillations in that time window.

gigaanalysis.diglock.**scanning_fft**(*signal*, *fs*, *tseg*, *tstep*, *nfft=None*, *window='hamming'*, *fit_point=5*, *low_f_skip=100*, *tqdm_bar=None*)

Finds the changing dominate frequency of a oscillatory signal.

Finds how the frequency of a oscillatory signal changes with time. This is achieved by performing many FFTs over a small window of signal which is slid along the complete signal. This is useful for extracting the measurement from PDO and TDO experiments.

Parameters

- **signal** (*numpy.ndarray*) – The data to extract the signal from in the form of a 1d array.
- **fs** (*float*) – The sample frequency of the measurement signal in Hertz.

- **tseg** (*float*) – The length in time to examine for each FFT in seconds.
- **tstep** (*float*) – How far to shift the window between each FFT in seconds.
- **nfft** (*None, optional*) – The number of points to use for the FFT extra points will be zero padded. The number of points used by default is `20*tseg*fs`, where `tseg*fs` is the length of the unpadded signal.
- **window** (*str, optional*) – The windowing function to used for the FFT that will be passed to `scipy.signal.get_window()`. THe default is 'hamming'.
- **fit_points** (*int, optional*) – The number of points to fit a parabola to identify the peak of the FFT. THe default is *5* and this is passed to *polypeak()*.
- **low_f_slip** (*int, optional*) – The number of points to skip when identifying the peak at the beginning of the FFT to ignore the low freq upturn. The default is *100* and this is passed to *polypeak()*.
- **tqdm_bar** (*tqdm.tqdm*, optional) – This function can be slow so a tqdm progress bar can be passed using this keyword which will be updated to show the progress of the calculation. This is done by:

```
from tqdm import tqdm
with tqdm() as bar:
    res = scanning_fft(signal, fs, tseg, tstep, tqdm_bar=bar)
```

Returns

- **times** (*numpy.ndarray*) – The midpoint of the time windows which the FFTs where taken at in seconds.
- **freqs** (*numpy.ndarray*) – The frequencies of the dominate oscillatory signal against time in Hertz.
- **amps** (*numpy.ndarray*) – The amplitude of the oscillatory signal from the FFT. This should be in the units of the signal.

gigaanalysis.diglock.**select_not_spikes**(*data*, *sdl=2.0*, *region=1001*)

Identifies spikes in the data and returns a boolean array.

This finds spikes in a set of data an excludes the region around them too. It does this by looking at where the value changes unusually quickly.

Parameters

- **data** (*numpy.ndarray*) – The signal to check for spikes.
- **sdl** (*float, optional*) – The number of standard deviations the data need to deviate by to be considered an outlier.
- **region** (*int, optional*) – The number of points around an outlier to exclude. Default is 1001.

Returns

good_vals (*numpy.ndarray*) – A boolean array with the same shape as the signal data with points near spikes labelled *False* and the unaffected points labelled *True*.

gigaanalysis.diglock.**spike_lock_in**(*signal*, *time_const*, *fs*, *freq*, *phase*, *sdl=2.0*, *region=1001*)

Performs a lock in of the signal but with also spike removal.

This lock in makes use of *ham_lock_in()*. It also removes the points effected by spikes by using *select_not_spikes()*. It tries to interpolate between the points. The spike removal works better with a smaller time constant.

Parameters

- **signal** (*numpy.ndarray*) – The AC signal to lock in to.
- **time_const** (*float*) – The time for averaging with the hamming window.
- **fs** (*float*) – The sample frequency.
- **freq** (*float*) – The frequency of the AC signal.
- **phase** (*float*) – The phase of the AC signal in degrees.
- **sdl** (*float, optional*) – The number of standard deviations that will trigger a spike detection. The default is 2.

Returns

locked_signal (*numpy.ndarray*) – The signal after the spike removal and lock in process.

2.13 GigaAnalysis - High Field - gigaanalysis.highfield

This program has a series of useful tools for conducting experiments in certain high field labs.

gigaanalysis.highfield.**PUtoB**(*PU_signal*, *field_factor*, *fit_points*, *to_fit='PU'*)

Converts the voltage from the pick up coil to field.

This is used for pulsed field measurements, where to obtain the filed the induced voltage in a coil is integrated. A fit is also applied because slight differences in the grounding voltage can cause a large change in the field so this needs to be corrected for.

Parameters

- **PU_signal** (*numpy.ndarray, Data*) – The signal from pick up coil.
- **field_factor** (*float*) – Factor to convert integral to magnetic field. Bare in mind this will change if the acquisition rate changes, for the same coil.
- **fit_points** (*int*) – Number of point at each end to remove offset.
- **to_fit** (*{'PU', 'field'} optional*) – If to correct an offset voltage the PU signal is fit or the field.

Returns

field (*numpy.ndarray, Data*) – An array of magnetic field the same length as PU_signal. If a *Data* is given then the y values are processed and a *Data* is returned.

class gigaanalysis.highfield.**PulsedLockIn**(*field*, *current*, *voltage*, *sample_freq=15000000.0*, *R_shunt=100.0*, *preamp=1.0*, *skip_num=200*, *B_min=0.0*, *side='down'*)

Bases: object

Performs a digital lock in on pulse field magnetotransport data.

This class is used to process data from pulsed field measurements using digital phase sensitive detection. The class is designed and named for it to be used for magnetotransport measurements, it can and has also been used for other types of experiments such as torque magnetometry. The type lock-in process it uses is convolution with a Hamming window.

As well as the simple phase sensitive detection functionality it also has tools for finding the phase shift, smoothing signal, and filtering out voltage spikes. The data produces can be accessed from the attributes or output as a *Data* object using one of the methods.

Parameters

- **field** (*numpy.ndarray*) – Field values in Tesla sorted in a 1D numpy array. The field will be changed to positive field is a negative sweep is given.
- **current** (*numpy.ndarray*) – Voltage reading across a shunt resistor in the form of a 1D numpy array.
- **voltage** (*numpy.ndarray*) – Voltage readings in the measurement channel in the form of a 1D numpy array.
- **sample_freq** (*float, optional*) – The rate of the data acquisition in Hertz. The default is *15e6* which is a common pulse field sample frequency.
- **R_shunt** (*float, optional*) – Value of the shunt resistor to measure the current in Ohms. The default value is *100*.
- **preamp** (*float, optional*) – Value of the amplification of the voltage signal before being measured. The default is *1*, so assumes no amplification.
- **skip_num** (*int, optional*) – The ratio of points to skip when outputting the data. This is used because the object sizes can become unwieldy if all the data is saved. The default is *200*, which drops 199 points for every one it keeps.
- **B_min** (*float or None, optional*) – The minimum value of the field to keep points lower in field to this will be dropped. If set to *None* all of the field range is kept. The default value is *0*, which only drops negative field values.
- **side** (*{'up', 'down', 'both'}, optional*) – The side of the pulse to produce the data for. 'up' is the first side, 'down' is the second, and 'both' takes both sides. The default is 'down'.

Variables

- **field** (*numpy.ndarray*) – The given numpy array containing the field values, if this is a negative field pulse then the sign of the field values are inverted.
- **Iv** (*numpy.ndarray*) – The numpy array given for the measurement current voltage.
- **Volt** (*numpy.ndarray*) – The numpy array given for the measurement voltage.
- **time** (*numpy.ndarray*) – The time values in milliseconds the same size as the given arrays.
- **fs** (*float*) – The sample frequency given in Hertz.
- **R_shunt** (*float*) – The given shunt voltage which is used to converted the measurement current voltage into current.
- **preamp** (*float*) – The given amplification that is used to convert the measured voltage into the generated voltage.
- **field_direction** (*bool*) – *True* if it is a positive pulse, *False* if it is a negative pulse.
- **peak_field** (*float*) – The maximum field value reached in the magnet pulse.
- **slice** (*slice*) – The slice that selects the data of interest out of the complete measurement. This is set by the *B_min*, *side*, and *step_size* keywords.
- **freq** (*float*) – The frequency of the applied measurement current voltage.
- **phase** (*float*) – The phase shift from the start of the file of the measurement current voltage in degrees.
- **Irms** (*float*) – The average applied current in root mean squared Amps.
- **time_const** (*float*) – The given time constant used for the voltage lock in seconds.
- **phase_shift** (*float*) – The given phase shift between the current measurement voltage and the experiment measurement voltage in degrees.

- **loc_Volt** (*numpy.ndarray*) – The experimental voltage after the lock in process considering the amplification in root mean squared Volts.
- **loc_Volt_out** (*numpy.ndarray*) – Equivalent to *loc_Volt* but for the out of phase component of the experimental voltage.
- **loc_I** (*numpy.ndarray*) – The applied current after a lock in process in root mean squared Amps.

auto_phase(*aim='change'*)

Finds the value of the phase_shift to achieve a certain result.

This finds the phase which makes the out of phase a flat as possible and also has the in phase be majority positive. It can also be set to move the majority of the signal into the in-phase channel using the *aim* parameter. Uses *diglock.phase_in()*.

Parameters

aim (*{'change', 'value'}*) – Weather to minimise the change in signal or the signal total in the out of phase channel.

Returns

phase (*float*) – A value between 0 and 360 which produces which most achieves the set goal in degrees.

current_in(*as_Data=True*, *x_axis='field'*)

The locked in current signal in units of Amps rms.

Parameters

- **as_Data** (*bool, optional*) – If *True*, which is the default, the data is returned as a *Data* object. If *False* it is returned as a numpy.ndarray.
- **x_axis** (*{'field', 'time'}*) – For the *Data* object whether the independent variable should be the applied field or the time. The default is the field.

Returns

current_in (*Data, numpy.ndarray*) – The locked in current signal in units of Amps rms.

find_phase(*skip_num=10*, *start_auto='change'*, *to_zero=False*)

Returns a function that makes a graph for phasing.

This produces a function which when called plots a graph showing the in and out of phase signal, the one argument is the phase. The default value of the one argument is set by the start_phase argument.

One way to use this is with the library ipywidgets which can make a slider in notebooks by running

```
find_phase_function = PulsedLockIn.find_phase()
ipywidgets.interact(find_phase_function,
   phase=ipywidgets.FloatSlider(min=0, max=360, step=1))
```

Parameters

- **skip_num** (*int, optional*) – The ratio of points to skip when plotting. As this requires a lot of calculation it can be beneficial to only plot a fraction speed up the process. The default value is 10.
- **start_auto** (*int, float, {'change', 'value'}, optional*) – Decides in what phase to start the graph at. If a string is given it is passed to *auto_phase()*. If an number is given the phase is set to that value.
- **to_zero** (*bool*) – If *True* the in phase and out of phase components are set to zero at the lowest field. This can make the changes easier to inspect. The default is *False*.

Returns

plotting (*Calculable*) – A function with one keyword argument of *phase* which plots the in and out of phase signal when called.

lockin_Volt(*time_const*, *phase_shift=None*)

This preforms a lock in process on the measurement signal.

This method sets the attributes `loc_Volt` and `loc_Volt_out`. The lock in process is performed using *diglock.ham_lock_in()*. Does not return anything.

Parameters

- **time_const** (*float*) – Time for averaging in seconds.
- **phase_shift** (*float, optional*) – Phase difference between the current voltage and the measurement voltage, this defaults to the attribute `phase_shift`. This is in degrees.

lockin_current(*time_const*, *phase=0*)

This preforms a lock in process on the current signal.

This uses *diglock.ham_lock_in()* to perform the lock in and sets the attribute `loc_I`.

Parameters

- **time_const** (*float*) – Time for averaging in seconds.
- **phase** (*float, optional*) – An applied phase shift for the current, in degrees. The default value is 0.

rephase(*phase_shift*, *trial=False*)

Rephases the signal to the new phase.

This changes the attributes `loc_Volt` and `loc_Volt_out` to shift the phase by a certain amount. If the *trial* is set to *True* then the result is returned instead of updating the attributes. The phase shift given is absolute and `phase_shft` is also updated.

Parameters

- **phase_shift** (*float*) – The new phase shift to use in degrees. This is absolute so the result is independent to the current phase shift.
- **trial** (*bool, optional*) – Whether to keep the new rephasing or to return the result instead. The default value is *False* which updates the attributes and returns nothing.

res_in(*as_Data=True*, *x_axis='field'*)

The locked in voltage signal, in phase in units of Ohms.

Parameters

- **as_Data** (*bool, optional*) – If *True*, which is the default, the data is returned as a *Data* object. If *False* it is returned as a `numpy.ndarray`.
- **x_axis** (*{'field', 'time'}*) – For the *Data* object whether the independent variable should be the applied field or the time. The default is the field.

Returns

res_in (*Data, numpy.ndarray*) – The locked in voltage signal, from the in phase channel divided by the average current to obtain the units in Ohms.

res_out(*as_Data=True*, *x_axis='field'*)

The locked in voltage signal, out of phase in units of Ohms.

Parameters

- **as_Data** (*bool, optional*) – If *True*, which is the default, the data is returned as a *Data* object. If *False* it is returned as a numpy.ndarray.
- **x_axis** (*{'field', 'time'}*) – For the *Data* object whether the independent variable should be the applied field or the time. The default is the field.

Returns

res_out (*Data, numpy.ndarray*) – The locked in voltage signal, from the out of phase channel divided by the average current to obtain the units in Ohms.

reset_slice(*skip_num='No'*, *B_min='No'*, *side='No'*, *trial=False*)

This reproduces the slice which selects the data of interest.

This is used to change the attribute `slice`. It also has a trial option that will return a new slice instead of updating the existing attribute. The parameters will try to all default to the values to reproduce the current slice, this maybe not be exacltly the same with *B_min*.

Parameters

- **skip_num** (*int, optional*) – The ratio of points to skip when outputting the data. This is used because the object sizes can become unwieldy if all the data is saved. it keeps.
- **B_min** (*float or None, optional*) – The minimum value of the field to keep points lower in field to this will be dropped. If set to *None* all of the field range is kept.
- **side** (*{'up', 'down', 'both'}, optional*) – The side of the pulse to produce the data for. 'up' is the first side, 'down' is the second, and 'both' takes both sides.
- **trial** (*bool, optional*) – If *True* the slice is not saved and instead returned. The default is *False* which updates `slice`.

smooth_Volts(*smooth_time*, *smooth_order=2*)

This smooths the measurement signal.

This must be applied after the lock in process. It changes the attributes `loc_Volt` and `loc_Volt_out`. The smoothing is done with a pass of a Savitzky-Golay filter from scipy.signal.savgol_filter(). This is particularly useful to remove small aliasing issues that can arise when using a short lock in window.

Parameters

- **smooth_points** (*float*) – The time window to fit the polynomial for smoothing, in seconds.
- **smooth_order** (*int, optional*) – The order of the poly to fit for the smoothing, the default is 2.

spike_lockin_Volt(*time_const*, *phase_shift=None*, *sdl=2*, *region=1001*)

Spike removing lock in process applied to the measurement signal.

This preforms a lock in process on the measurement signal, with the aim of removing spikes in the raw first. This can be useful as some magnets can see high voltage spikes. This uses *diglock.spike_lock_in()*. Nothing is returned but the following attributes are updated, `time_const`, `phase_shift`, `loc_Volt`, and `loc_Volt_out`.

Parameters

- **time_const** (*float*) – Time for averaging in seconds.
- **phase_shift** (*float, optional*) – Phase difference between the current voltage and the measurement voltage in degrees. This defaults to the attribute `phase_shift`.
- **sdl** (*float, optional*) – The number of standard deviations the data to be deviate from to be considered a outlier. Outliers are identified as spikes. The default is 2.

- **region** (*int, optional*) – The number of points around the outlier that are also considered compromised. The default is 1001.

volts_in(*as_Data=True*, *x_axis='field'*)

The signal from the locked in, in phase voltage.

Parameters

- **as_Data** (*bool, optional*) – If *True*, which is the default, the data is returned as a *Data* object. If *False* it is returned as a numpy.ndarray.
- **x_axis** (*{'field', 'time'}*) – For the *Data* object whether the independent variable should be the applied field or the time. The default is the field.

Returns

volts_in (*Data, numpy.ndarray*) – The locked in measurement signal from the in phase channel.

volts_out(*as_Data=True*, *x_axis='field'*)

The signal from the locked in, out of phase voltage.

Parameters

- **as_Data** (*bool, optional*) – If *True*, which is the default, the data is returned as a *Data* object. If *False* it is returned as a numpy.ndarray.
- **x_axis** (*{'field', 'time'}*) – For the *Data* object whether the independent variable should be the applied field or the time. The default is the field.

Returns

volts_out (*Data, numpy.ndarray*) – The locked in measurement signal from the out of phase channel.

volts_over_current(*as_Data=True*, *x_axis='field'*)

The locked in voltage signal over the current signal.

The is for the same purpose as *res_in()* but if the applied current is for some reason not stable.

Parameters

- **as_Data** (*bool, optional*) – If *True*, which is the default, the data is returned as a *Data* object. If *False* it is returned as a numpy.ndarray.
- **x_axis** (*{'field', 'time'}*) – For the *Data* object whether the independent variable should be the applied field or the time. The default is the field.

Returns

res_in (*Data, numpy.ndarray*) – The locked in voltage signal from the in phase channel divided by the locked in current signal. This also obtains the value in units of Ohms but allows to take into consideration variable current flow.

gigaanalysis.highfield.**example_field**(*max_field*, *peak_time*, *length*, *sample_rate*, *as_Data=False*)

Produces a data set with a field profile.

This field profile matches the KS3 magnet in HLD. The pulse time and maximum field has been normalised, and is get by the user. Originally they where 68.9 Tesla and 0.0336 seconds. I find this useful for simulating test measurements.

Parameters

- **max_field** (*float*) – The maximum field value that the data will reach.
- **peak_time** (*float*) – The time that the magnet reaches peak field.

- **length** (*int*) – The number of data points in the file.
- **sample_rate** (*float*) – The sample frequency of the data.
- **as_Data** (*bool, optional*) – If true a *Data* class is returned.

Returns

field_profile (*numpy.ndarray or Data*) – The field values simulated for the parameters given.

gigaanalysis.highfield.**pick_pulse_side**(*field*, *B_min*, *side*, *skip_num=1*, *give_slice=True*)

Produces a slice that selects a certain section of a filed pulse.

This takes a field profile and produces a slice that has one side or both in.

Parameters

- **field** (*numpy.ndarray*) – Field values in a 1D numpy array, the field needs to be pulsed in the positive direction. If you want to analyse a negative sweep first take the negative of it.
- **B_min** (*float or None*) – The value of field to cut all the lower data off. This is used as sometimes the tails of the pulses can be very long. If it is set to None the full range is kept.
- **side** (*{'up', 'down', 'both'}*) – Which side of the the pulse to take. 'up' takes the first side, 'down' takes the second, and 'both' includes both sides of the pulse.
- **skip_num** (*int, optional*) – The ratio of points to skip to reduce the size of the data set. The default is *1*, which doesn't slip any points.
- **give_slice** (*bool, optional*) – If the default of *True* a slice is returned as described. If *False* then the field is returned with the slice applied to it.

Returns

B_slice (*slice, numpy.ndarray*) – The slice to apply to take one field side. If *give_slice* is *False* then the filed array is returned with the slice applied.

2.14 GigaAnalysis - Constants - gigaanalysis.const

Here is contained a collection of functions with when called return values of physical constants. They always return floats and all have one optional parameter 'unit' which default is 'SI' for the International System of Units values for these parameters. The module scipy.constants contains many more than what is listed here, but I included these for the different units.

gigaanalysis.const.**G**(*unit='SI'*)

Gravitational constant

Unit	Value
'SI'	6.67430e-11 m^3/kg/s^2
'CGS'	6.67430e-8 dyn cm^2/g^2

Parameters

unit (*str, optional*) – The unit system to give the value in.

Returns

Value of the Gravitational constant (*float*)

gigaanalysis.const.**Na**(*unit='SI'*)

Avogadro constant

Unit	Value
'SI'	6.02214076e+23 1/mol

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Avogadro constant (*float*)

gigaanalysis.const.**R**(*unit='SI'*)

Gas Constant

Unit	Value
'SI'	8.31446261815324 J/K/mol
'eV'	5.189479388046824e+19 eV/K/mol
'CGS'	8.31446261815324e+7 erg/K/mol

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Gas Constant (*float*)

gigaanalysis.const.**a0**(*unit='SI'*)

Bohr radius

Unit	Value
'SI'	5.29177210903e-11 m
'CGS'	5.29177210903e-9 cm

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Bohr radius (*float*)

gigaanalysis.const.**alpha**(*unit='SI'*)

Fine-structure constant

Unit	Value
'SI'	7.2973525693e-3

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Fine-structure constant (*float*)

gigaanalysis.const.**amu**(*unit='SI'*)

Unified Atomic mass unit or Dalton

Unit	Value
'SI'	1.66053906660e-27 kg
'CGS'	1.66053906660e-24

Parameters

unit (*str, optional*) – The unit system to give the value in.

Returns

Value of the Atomic mass unit (*float*)

gigaanalysis.const.**c**(*unit='SI'*)

Speed of light

Unit	Value
'SI'	2.99792458e+8 m/s
'CGS'	2.99792458e+10 cm/s

Parameters

unit (*str, optional*) – The unit system to give the value in.

Returns

Value of the speed of light (*float*)

gigaanalysis.const.**ep0**(*unit='SI'*)

Vacuum permittivity

Unit	Value
'SI'	8.8541878128e-12 F/m
'eV'	1.4185972826e-30 C^2/eV

Parameters

unit (*str, optional*) – The unit system to give the value in.

Returns

Value of the Vacuum permittivity (*float*)

gigaanalysis.const.**h**(*unit='SI'*)

Planck constant

Unit	Value
'SI'	6.62607015e-34 J s
'eV'	4.135667696e-15 eV s
'CGS'	6.62607015e-27 erg s

Parameters

unit (*str, optional*) – The unit system to give the value in.

Returns

Value of the Planck constant (*float*)

gigaanalysis.const.**hbar**(*unit='SI'*)

Reduced Planck constant

Unit	Value
'SI'	1.054571817e-34 J s
'eV'	6.582119569e-16 eV s
'CGS'	1.054571817e-27 erg s

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Reduced Planck constant (*float*)

gigaanalysis.const.**kb**(*unit='SI'*)

Boltzmann constant

Unit	Value
'SI'	1.380649e-23 J/K
'eV'	8.617333262145e-5 eV/K
'CGS'	1.380649e-16 erg/K/

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Boltzmann constant (*float*)

gigaanalysis.const.**me**(*unit='SI'*)

Electron rest mass

Unit	Value
'SI'	9.1093837015e-31 kg
'CGS'	9.1093837015e-29 g
'MeVc'	5.1099895000e-1 MeV/c^2
'uamu'	5.48579909065e-4 Da

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Bohr magneton (*float*)

gigaanalysis.const.**mp**(*unit='SI'*)

Proton rest mass

Unit	Value
'SI'	1.67262192369e-27 kg
'CGS'	1.67262192369e-25 g
'MeVc'	9.3827208816e+2 MeV/c^2
'uamu'	1.007276466621e+0 Da

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Nuclear magneton (*float*)

gigaanalysis.const.**mu0**(*unit='SI'*)

Vacuum permeability

Unit	Value
'SI'	1.25663706212e-6 H/m
'eV'	7.8433116265e+12 eV/A^2

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Vacuum permeability (*float*)

gigaanalysis.const.**muB**(*unit='SI'*)

Bohr magneton

Unit	Value
'SI'	9.274009994e-24 J/T
'eV'	5.7883818012e-5 eV/T
'CGS'	9.274009994e-21 erg/T

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Bohr magneton (*float*)

gigaanalysis.const.**muN**(*unit='SI'*)

Nuclear magneton

Unit	Value
'SI'	5.050783699e-27 J/T
'eV'	3.1524512550e-8 eV/T
'CGS'	5.050783699e-24 erg/T

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Nuclear magneton (*float*)

gigaanalysis.const.**qe**(*unit='SI'*)

Elementary charge

Unit	Value
'SI'	1.602176634e-19 C
'CGS'	1.602176634e-20 statC

Parameters
unit (*str, optional*) – The unit system to give the value in.

Returns
Value of the Elementary charge (*float*)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

X

Y