# optoanalysis Documentation

***Release 4.2.7***

**Ashley Setter** **Markus Rademacher**

**May 13, 2021**

# Contents:

This is the documentation for the optoanalysis package developed primarily by Ashley Setter of the Quantum Nanophysics and Matter Wave Interferometry group headed up by Prof. Hendrik Ulbricht at Southampton University in the UK.

The thermo module of this package was developed mainly by Markus Rademacher of the University of Vienna in Austria, who works in the group of Markus Aspelmeyer and Nikolai Kiesel.

This library contains numerous functions for loading, analysing and plotting data produced from our optically levitated nanoparticle experiment. We use an optical tweezer setup to optically trap and levitate nanoparticles in intense laser light and measure the motion of these particles interferometrically from the light they scatter. This library provides all the tools to load up examples of this kind of data and analyse it. Currently data can be loaded from .trc or .raw binary files produced by Teledyne LeCroy oscilloscopes and .bin files produced from by Saleae data loggers.

If you use this package in any academic work it would be very appretiated if you could cite it.

CHAPTER 1

# optoanalysis package

## 1.1 optoanalysis.optoanalysis module

**class** optoanalysis.optoanalysis.**DataObject**(*filepath*, *RelativeChannelNo=None*, *SampleFreq=None*, *NumberOfChannels=None*, *PointsToLoad=-1*, *calcPSD=True*, *NPerSegmentPSD=1000000*, *NormaliseByMonitorOutput=False*)

Bases: object

Creates an object containing data and all it's properties.

**Attributes**

**filepath** [string] filepath to the file containing the data used to initialise this particular instance of the DataObject class

**filename** [string] filename of the file containing the data used to initialise this particular instance of the DataObject class

**time** [frange] Contains the time data as an frange object. Can get a generator or array of this object.

**voltage** [ndarray] Contains the voltage data in Volts

**SampleFreq** [sample frequency used to sample the data (when it was] taken by the oscilloscope)

**freqs** [ndarray] Contains the frequencies corresponding to the PSD (Pulse Spectral Density)

**PSD** [ndarray] Contains the values for the PSD (Pulse Spectral Density) as calculated at each frequency contained in freqs

**calc_area_under_PSD**(*lowerFreq*, *upperFreq*)

Sums the area under the PSD from lowerFreq to upperFreq.

**Parameters**

**lowerFreq** [float] The lower limit of frequency to sum from

**upperFreq** [float] The upper limit of frequency to sum to

**Returns**

**AreaUnderPSD** [float] The area under the PSD from lowerFreq to upperFreq

**calc_gamma_from_RSquaredPSD_fit**(*GammaGuess=None*, *CutOffFreq=None*, *FreqTrapGuess=None*, *AGuess=None*, *OffsetGuess=None*, *FractionOfSampleFreq=1*, *NPerSegmentPSD=None*, *Fit_xlim=None*, *silent=False*, *MakeFig=True*, *show_fig=True*)

Calculates the total damping, i.e. Gamma, by calculating the RSquared PSD of the position-time trace. The RSquared is fitted with the R^2 function. The methodology is explained in the following paper (DOI: 10.1103/PhysRevResearch.2.023349) and the function returns the parameters with errors.

**Parameters**

**GammaGuess** [float, optional] Inital guess for BigGamma in Radians If None takes Gamma from a previously done PSD fit

**CutOffFreq** [float, optional] is the cut off frequncy to get rid of the 2*omega component, make this several times larger than your linewidth, in Hz

**FreqTrapGuess** [float, optional] Inital guess for the trapping Frequency in Hz If None takes FreqTrap from a previously done PSD fit

**AGuess** [float, optional] Inital guess for the multiplicative factor in R^2 which equals: 8*(S_F/(2m^2*Omega0^2))^2 If None, AGuess set to 1.

**OffsetGuess** [float, optional] Additive Offset to the fitting equation. If None, OffsetGuess is set to 0.

**FractionOfSampleFreq** [integer, optional] The fraction of the sample frequency to subsample the data by. This sometimes needs to be done because a filter with the appropriate frequency response may not be generated using the sample rate at which the data was taken. Increasing this number means the R Squared signal produced by this function will be sampled at a lower rate but a higher number means a higher chance that the filter produced will have a nice frequency response.

**NPerSegmentPSD** [int, optional] NPerSegment to pass to scipy.signal.welch to calculate the PSD default = 100/BigGamma*SampleFreq

**Fit_xlim** [list of float, optional] limits the R Squared PSD signal used for the fit function, i.e.: [lowerLimit, upperLimit] default = [RSquared_freqs[0], 0.75 * CutOffFreq]

**silent** [bool, optional] Whether it prints the values fitted or is silent.

**MakeFig** [bool, optional] Whether to construct and return the figure object showing the fitting. defaults to True

**show_fig** [bool, optional] Whether to show the figure object when it has been created. defaults to True

**Returns**

**Gamma** [ufloat] Big Gamma, the total damping in radians

**fig** [matplotlib.figure.Figure object] The figure object created showing the autocorrelation of the data with the fit

**ax** [matplotlib.axes.Axes object] The axes object created showing the autocorrelation of the data with the fit

**calc_gamma_from_energy_autocorrelation_fit**(*GammaGuess=None*, *silent=False*, *MakeFig=True*, *show_fig=True*)

Calculates the total damping, i.e. Gamma, by calculating the energy each point in time. This energy array is then used for the autocorrleation. The autocorrelation is fitted with an exponential relaxation function and the function returns the parameters with errors.

**Parameters**

**GammaGuess** [float, optional] Inital guess for BigGamma (in radians)

**silent** [bool, optional] Whether it prints the values fitted or is silent.

**MakeFig** [bool, optional] Whether to construct and return the figure object showing the fitting. defaults to True

**show_fig** [bool, optional] Whether to show the figure object when it has been created. defaults to True

**Returns**

**Gamma** [ufloat] Big Gamma, the total damping in radians

**fig** [matplotlib.figure.Figure object] The figure object created showing the autocorrelation of the data with the fit

**ax** [matplotlib.axes.Axes object] The axes object created showing the autocorrelation of the data with the fit

**calc_gamma_from_position_autocorrelation_fit**(*GammaGuess=None*, *FreqTrapGuess=None*, *silent=False*, *MakeFig=True*, *show_fig=True*)

Calculates the total damping, i.e. Gamma, by calculating the autocorrleation of the position-time trace. The autocorrelation is fitted with an exponential relaxation function derived in Tongcang Li's 2013 thesis (DOI: 10.1007/978-1-4614-6031-2) and the function (equation 4.20 in the thesis) returns the parameters with errors.

**Parameters**

**GammaGuess** [float, optional] Inital guess for BigGamma (in radians)

**FreqTrapGuess** [float, optional] Inital guess for the trapping Frequency in Hz

**silent** [bool, optional] Whether it prints the values fitted or is silent.

**MakeFig** [bool, optional] Whether to construct and return the figure object showing the fitting. defaults to True

**show_fig** [bool, optional] Whether to show the figure object when it has been created. defaults to True

**Returns**

**Gamma** [ufloat] Big Gamma, the total damping in radians

**OmegaTrap** [ufloat] Trapping frequency in radians

**fig** [matplotlib.figure.Figure object] The figure object created showing the autocorrelation of the data with the fit

**ax** [matplotlib.axes.Axes object] The axes object created showing the autocorrelation of the data with the fit

**calc_gamma_from_variance_autocorrelation_fit**(*NumberOfOscillations*, *GammaGuess=None*, *silent=False*, *MakeFig=True*, *show_fig=True*)

Calculates the total damping, i.e. Gamma, by splitting the time trace into chunks of NumberOfOscillations oscillations and calculated the variance of each of these chunks. This array of varainces is then used for the autocorrleation. The autocorrelation is fitted with an exponential relaxation function and the function returns the parameters with errors.

**Parameters**

**NumberOfOscillations** [int] The number of oscillations each chunk of the timetrace used to calculate the variance should contain.

**GammaGuess** [float, optional] Inital guess for BigGamma (in radians)

**Silent** [bool, optional] Whether it prints the values fitted or is silent.

**MakeFig** [bool, optional] Whether to construct and return the figure object showing the fitting. defaults to True

**show_fig** [bool, optional] Whether to show the figure object when it has been created. defaults to True

**Returns**

**Gamma** [ufloat] Big Gamma, the total damping in radians

**fig** [matplotlib.figure.Figure object] The figure object created showing the autocorrelation of the data with the fit

**ax** [matplotlib.axes.Axes object] The axes object created showing the autocorrelation of the data with the fit

**calc_phase_space**(*freq*, *ConvFactor*, *PeakWidth=10000*, *FractionOfSampleFreq=1*, *timeStart=None*, *timeEnd=None*, *PointsOfPadding=500*, *ShowPSD=False*)

Calculates the position and velocity (in m) for use in plotting the phase space distribution.

**Parameters**

**freq** [float] The frequenecy of the peak (Trapping frequency of the dimension of interest)

**ConvFactor** [float (or ufloat)] The conversion factor between Volts and Meters

**PeakWidth** [float, optional] The width of the peak. Defaults to 10KHz

**FractionOfSampleFreq** [int, optional] The fraction of the sample freq to use to filter the data. Defaults to 1.

**timeStart** [float, optional] Starting time for data from which to calculate the phase space. Defaults to start of time data.

**timeEnd** [float, optional] Ending time for data from which to calculate the phase space. Defaults to start of time data.

**PointsOfPadding** [float, optional] How many points of the data at the beginning and end to disregard for plotting the phase space, to remove filtering artifacts. Defaults to 500

**ShowPSD** [bool, optional] Where to show the PSD of the unfiltered and the filtered signal used to make the phase space plot. Defaults to False.

***args, **kwargs** [optional] args and kwargs passed to qplots.joint_plot

**Returns**

**time** [ndarray] time corresponding to position and velocity

**PosArray** [ndarray] Array of position of the particle in time

**VelArray** [ndarray] Array of velocity of the particle in time

**extract_ZXY_motion**(*ApproxZXYFreqs*, *uncertaintyInFreqs*, *ZXYPeakWidths*, *subSampleFraction=1*, *NPerSegmentPSD=1000000*, *MakeFig=True*, *show_fig=True*)

Extracts the x, y and z signals (in volts) from the voltage signal. Does this by finding the highest peaks in the signal about the approximate frequencies, using the uncertaintyinfreqs parameter as the width it searches. It then uses the ZXYPeakWidths to construct bandpass IIR filters for each frequency and filtering them. If too high a sample frequency has been used to collect the data scipy may not be able to construct a filter good enough, in this case increasing the subSampleFraction may be nessesary.

**Parameters**

**ApproxZXYFreqs** [array_like] A sequency containing 3 elements, the approximate z, x and y frequency respectively.

**uncertaintyInFreqs** [float] The uncertainty in the z, x and y frequency respectively.

**ZXYPeakWidths** [array_like] A sequency containing 3 elements, the widths of the z, x and y frequency peaks respectively.

**subSampleFraction** [int, optional] How much to sub-sample the data by before filtering, effectively reducing the sample frequency by this fraction.

**NPerSegmentPSD** [int, optional] NPerSegment to pass to scipy.signal.welch to calculate the PSD

**show_fig** [bool, optional] Whether to show the figures produced of the PSD of the original signal along with the filtered x, y and z.

**Returns**

**self.zVolts** [ndarray] The z signal in volts extracted by bandpass IIR filtering

**self.xVolts** [ndarray] The x signal in volts extracted by bandpass IIR filtering

**self.yVolts** [ndarray] The y signal in volts extracted by bandpass IIR filtering

**time** [ndarray] The array of times corresponding to the above 3 arrays

**fig** [matplotlib.figure.Figure object] figure object containing a plot of the PSD of the original signal with the z, x and y filtered signals

**ax** [matplotlib.axes.Axes object] axes object corresponding to the above figure

**extract_parameters**(*P_mbar*, *P_Error*, *method='chang'*)

Extracts the Radius, mass and Conversion factor for a particle.

**Parameters**

**P_mbar** [float] The pressure in mbar when the data was taken.

**P_Error** [float] The error in the pressure value (as a decimal e.g. 15% = 0.15)

**Returns**

**Radius** [uncertainties.ufloat] The radius of the particle in m

**Mass** [uncertainties.ufloat] The mass of the particle in kg

**ConvFactor** [uncertainties.ufloat] The conversion factor between volts/m

**filter_data**(*freq*, *FractionOfSampleFreq=1*, *PeakWidth=10000*, *filterImplementation='filtfilt'*, *timeStart=None*, *timeEnd=None*, *NPerSegmentPSD=1000000*, *PyCUDA=False*, *MakeFig=True*, *show_fig=True*)

filter out data about a central frequency with some bandwidth using an IIR filter.

**Parameters**

**freq** [float] The frequency of the peak of interest in the PSD

**FractionOfSampleFreq** [integer, optional] The fraction of the sample frequency to sub-sample the data by. This sometimes needs to be done because a filter with the appropriate frequency response may not be generated using the sample rate at which the data was taken. Increasing this number means the x, y and z signals produced by this function will be sampled at a lower rate but a higher number means a higher chance that the filter produced will have a nice frequency response.

**PeakWidth** [float, optional] The width of the pass-band of the IIR filter to be generated to filter the peak. Defaults to 10KHz

**filterImplementation** [string, optional] filtfilt or lfilter - use scipy.filtfilt or lfilter ifft - uses built in IFFT_filter default: filtfilt

**timeStart** [float, optional] Starting time for filtering. Defaults to start of time data.

**timeEnd** [float, optional] Ending time for filtering. Defaults to end of time data.

**NPerSegmentPSD** [int, optional] NPerSegment to pass to scipy.signal.welch to calculate the PSD

**PyCUDA** [bool, optional] Only important for the 'ifft'-method If True, uses PyCUDA to accelerate the FFT and IFFT via using your NVIDIA-GPU If False, performs FFT and IFFT with conventional scipy.fftpack

**MakeFig** [bool, optional] If True - generate figure showing filtered and unfiltered PSD Defaults to True.

**show_fig** [bool, optional] If True - plot unfiltered and filtered PSD Defaults to True.

**Returns**

**timedata** [ndarray] Array containing the time data

**FiletedData** [ndarray] Array containing the filtered signal in volts with time.

**fig** [matplotlib.figure.Figure object] The figure object created showing the PSD of the filtered and unfiltered signal

**ax** [matplotlib.axes.Axes object] The axes object created showing the PSD of the filtered and unfiltered signal

**get_PSD** (*NPerSegment=1000000*, *window='hann'*, *timeStart=None*, *timeEnd=None*, *override=False*)
Extracts the power spectral density (PSD) from the data.

**Parameters**

**NPerSegment** [int, optional] Length of each segment used in scipy.welch default = 1000000

**window** [str or tuple or array_like, optional] Desired window to use. See get_window for a list of windows and required parameters. If window is array_like it will be used directly as the window and its length will be used for nperseg. default = "hann"

**Returns**

**freqs** [ndarray] Array containing the frequencies at which the PSD has been calculated

**PSD** [ndarray] Array containing the value of the PSD at the corresponding frequency value in V**2/Hz

**get_fit** (*TrapFreq*, *WidthOfPeakToFit*, *A_Initial=1000000000.0*, *Gamma_Initial=400*, *silent=False*, *MakeFig=True*, *show_fig=True*, *plot_initial=True*)
Function that fits to a peak to the PSD to extract the frequency, A factor and Gamma (damping) factor.

**Parameters**

**TrapFreq** [float] The approximate trapping frequency to use initially as the centre of the peak

**WidthOfPeakToFit** [float] The width of the peak to be fitted to. This limits the region that the fitting function can see in order to stop it from fitting to the wrong peak

**A_Initial** [float, optional] The initial value of the A parameter to use in fitting

**Gamma_Initial** [float, optional] The initial value of the Gamma parameter to use in fitting

**Silent** [bool, optional] Whether to print any output when running this function defaults to False

**MakeFig** [bool, optional] Whether to construct and return the figure object showing the fitting. defaults to True

**show_fig** [bool, optional] Whether to show the figure object when it has been created. defaults to True

**Returns**

**A** [uncertainties.ufloat] Fitting constant A A = $\gamma$**2*2*Γ_0*(K_b*T_0)/($\pi$*m) where: $\gamma$ = conversionFactor Γ_0 = Damping factor due to environment $\pi$ = pi

**OmegaTrap** [uncertainties.ufloat] The trapping frequency in the z axis (in angular frequency)

**Gamma** [uncertainties.ufloat] The damping factor Gamma = Γ = Γ_0 + $\delta$Γ where: Γ_0 = Damping factor due to environment $\delta$Γ = extra damping due to feedback or other effects

**fig** [matplotlib.figure.Figure object] figure object containing the plot

**ax** [matplotlib.axes.Axes object] axes with the data plotted of the: - initial data - smoothed data - initial fit - final fit

**get_fit_auto**(*CentralFreq*, *MaxWidth=15000*, *MinWidth=500*, *WidthIntervals=500*, *MakeFig=True*, *show_fig=True*, *silent=False*, *plot_initial=True*)

Tries a range of regions to search for peaks and runs the one with the least error and returns the parameters with the least errors.

**Parameters**

**CentralFreq** [float] The central frequency to use for the fittings.

**MaxWidth** [float, optional] The maximum bandwidth to use for the fitting of the peaks.

**MinWidth** [float, optional] The minimum bandwidth to use for the fitting of the peaks.

**WidthIntervals** [float, optional] The intervals to use in going between the MaxWidth and MinWidth.

**show_fig** [bool, optional] Whether to plot and show the final (best) fitting or not.

**Returns**

**OmegaTrap** [ufloat] Trapping frequency

**A** [ufloat] A parameter

**Gamma** [ufloat] Gamma, the damping parameter

**fig** [matplotlib.figure.Figure object] The figure object created showing the PSD of the data with the fit

**ax** [matplotlib.axes.Axes object] The axes object created showing the PSD of the data with the fit

**get_fit_from_peak**(*lowerLimit*, *upperLimit*, *NumPointsSmoothing=1*, *silent=False*, *MakeFig=True*, *show_fig=True*, *plot_initial=True*)

Finds approximate values for the peaks central frequency, height, and FWHM by looking for the heighest peak in the frequency range defined by the input arguments. It then uses the central frequency as the trapping frequency, peak height to approximate the A value and the FWHM to an approximate the Gamma (damping) value.

**Parameters**

**lowerLimit** [float] The lower frequency limit of the range in which it looks for a peak

**upperLimit** [float] The higher frequency limit of the range in which it looks for a peak

**NumPointsSmoothing** [float] The number of points of moving-average smoothing it applies before fitting the peak.

**Silent** [bool, optional] Whether it prints the values fitted or is silent.

**show_fig** [bool, optional] Whether it makes and shows the figure object or not.

**Returns**

**OmegaTrap** [ufloat] Trapping frequency

**A** [ufloat] A parameter

**Gamma** [ufloat] Gamma, the damping parameter

**get_time_data**(*timeStart=None*, *timeEnd=None*)

Gets the time and voltage data.

**Parameters**

**timeStart** [float, optional] The time get data from. By default it uses the first time point

**timeEnd** [float, optional] The time to finish getting data from. By default it uses the last time point

**Returns**

**time** [ndarray] array containing the value of time (in seconds) at which the voltage is sampled

**voltage** [ndarray] array containing the sampled voltages

**load_time_data**(*RelativeChannelNo=None*, *SampleFreq=None*, *NumberOfChannels=None*, *PointsToLoad=-1*, *NormaliseByMonitorOutput=False*)

Loads the time and voltage data and the wave description from the associated file.

**Parameters**

**RelativeChannelNo** [int, optional] Channel number for loading .bin saleae data files If loading a .mat file produced by the picoscope using picolog, used to specifiy the channel ID as follows: 0 = Channel ‘A’, 1 = Channel ‘B’, 2 = Channel ‘C’ and 3 = Channel ‘D’ If loading a .bin file saved using custom code to interface with the Picoscope used to specify the channel number to load in conjunction with the NumberOfChannels parameter, if left None with .bin files it will assume that the file to load only contains one channel. If loading a .dat file produced by the labview NI5122 daq card, used to specifiy the channel number if two channels where saved, if left None with .dat files it will assume that the file to load only contains one channel. If NormaliseByMonitorOutput is True then RelativeChannelNo specifies the monitor channel for loading a .dat file produced by the labview NI5122 daq card.

**SampleFreq** [float, optional] Manual selection of sample frequency for loading labview NI5122 daq files and .mat and .bin files recorded using the Picoscope

**NumberOfChannels** [int, optional] Total number of channels present in a .bin file recorded using a Picoscope.

**PointsToLoad** [int, optional] Number of first points to read. -1 means all points (i.e., the complete file) WORKS WITH NI5122 AND PICOSCOPE .BIN DATA SO FAR ONLY!!!

**NormaliseByMonitorOutput** [bool, optional] If True the particle signal trace will be divided by the monitor output, which is specified by the channel number set in the RelativeChannelNo parameter. WORKS WITH NI5122 DATA SO FAR ONLY!!!

**plot_PSD**(*xlim=None*, *units='kHz'*, *show_fig=True*, *timeStart=None*, *timeEnd=None*, **args*, ***kwargs*)

plot the pulse spectral density.

**Parameters**

**xlim** [array_like, optional] The x limits of the plotted PSD [LowerLimit, UpperLimit] Default value is [0, SampleFreq/2]

**units** [string, optional] Units of frequency to plot on the x axis - defaults to kHz

**show_fig** [bool, optional] If True runs plt.show() before returning figure if False it just returns the figure object. (the default is True, it shows the figure)

**Returns**

**fig** [matplotlib.figure.Figure object] The figure object created

**ax** [matplotlib.axes.Axes object] The subplot object created

**plot_phase_space**(*freq*, *ConvFactor*, *PeakWidth=10000*, *FractionOfSampleFreq=1*, *timeStart=None*, *timeEnd=None*, *PointsOfPadding=500*, *units='nm'*, *show_fig=True*, *ShowPSD=False*, *xlabel=''*, *ylabel=''*, **args*, ***kwargs*)

**plot_phase_space_sns**(*freq*, *ConvFactor*, *PeakWidth=10000*, *FractionOfSampleFreq=1*, *kind='hex'*, *timeStart=None*, *timeEnd=None*, *PointsOfPadding=500*, *units='nm'*, *logscale=False*, *cmap=None*, *marginalColor=None*, *gridsize=200*, *show_fig=True*, *ShowPSD=False*, *alpha=0.5*, **args*, ***kwargs*)

Plots the phase space of a peak in the PSD.

**Parameters**

**freq** [float] The frequenecy of the peak (Trapping frequency of the dimension of interest)

**ConvFactor** [float (or ufloat)] The conversion factor between Volts and Meters

**PeakWidth** [float, optional] The width of the peak. Defaults to 10KHz

**FractionOfSampleFreq** [int, optional] The fraction of the sample freq to use to filter the data. Defaults to 1.

**kind** [string, optional] kind of plot to draw - pass to jointplot from seaborne

**timeStart** [float, optional] Starting time for data from which to calculate the phase space. Defaults to start of time data.

**timeEnd** [float, optional] Ending time for data from which to calculate the phase space. Defaults to start of time data.

**PointsOfPadding** [float, optional] How many points of the data at the beginning and end to disregard for plotting the phase space, to remove filtering artifacts. Defaults to 500.

**units** [string, optional] Units of position to plot on the axis - defaults to nm

**cmap** [matplotlib.colors.ListedColormap, optional] cmap to use for plotting the jointplot

**marginalColor** [string, optional] color to use for marginal plots

**gridsize** [int, optional] size of the grid to use with kind="hex"

**show_fig** [bool, optional] Whether to show the figure before exiting the function Defaults to True.

**ShowPSD** [bool, optional] Where to show the PSD of the unfiltered and the filtered signal used to make the phase space plot. Defaults to False.

**Returns**

**fig** [matplotlib.figure.Figure object] figure object containing the phase space plot

**JP** [seaborn.jointplot object] joint plot object containing the phase space plot

**plot_spectrogram**(*timePerFFT=0.0003*, *title=''*, *ylim=None*, *timeStart=None*, *timeEnd=None*, *xunits='s'*, *yunits='kHz'*, *return_data=False*, *animate=False*, *filename='animation.gif'*, *show_fig=True*, ***kwargs*)

plot the spectrogram or produce an animated plot the spectrogram.

**Parameters**

**timePerFFT** [float, default: 1e-3] The time in xunits used in each block for the FFT.

**title** [string, optional] title to be displayed on the plot

**ylim** [array_like, optional] The y limits of the plotted spectrogram [LowerLimit, UpperLimit] Default value is [0, SampleFreq/2]

**timeStart** [float, optional] Starting time for spectrogram calculation. Defaults to start of time data.

**timeEnd** [float, optional] Ending time for spectrogram calculation. Defaults to end of time data.

**xunits** [string, optional] Units of time used for timePerFFT, timeStart and timeEnd - defaults to s

**yunits** [string, optional] Units of frequency limits to plot on the y axis - defaults to kHz

**return_data** [bool, optional] If True data (spec, freqs and t) of spectrogram will be returned

**animate** [bool, optional] If True will animate the spectrogram plot.

**filename** [string, optional] filename to save animation

**show_fig** [bool, optional] If True runs plt.show() before returning figure if False it just returns the figure object. (the default is True, it shows the figure)

**Returns**

**spectrum** [2D array] Columns are the periodograms of successive segments. Only returned if return_data=True

**freqs** [1-D array] The frequencies corresponding to the rows in *spectrum*. Only returned if return_data=True

**t** [1-D array] The times corresponding to midpoints of segments (i.e., the columns in *spectrum*). Only returned if return_data=True

**fig** [matplotlib.figure.Figure object] The figure object created

**ax** [matplotlib.axes.Axes object] The subplot object created

**plot_time_data**(*timeStart=None*, *timeEnd=None*, *units='s'*, *show_fig=True*)
plot time data against voltage data.

**Parameters**

**timeStart** [float, optional] The time to start plotting from. By default it uses the first time point

**timeEnd** [float, optional] The time to finish plotting at. By default it uses the last time point

**units** [string, optional] units of time to plot on the x axis - defaults to s

**show_fig** [bool, optional] If True runs plt.show() before returning figure if False it just returns the figure object. (the default is True, it shows the figure)

**Returns**

**fig** [matplotlib.figure.Figure object] The figure object created

**ax** [matplotlib.axes.Axes object] The subplot object created

**write_time_data**(*filename*)
Writes time data to a csv file.

**Parameters**

**filename** [string] filename of csv file to be written

optoanalysis.optoanalysis.**GenCmap**(*basecolor*, *ColorRange*, *NumOfColors*, *logscale=False*)

optoanalysis.optoanalysis.**IFFT_filter**(*Signal*, *SampleFreq*, *lowerFreq*, *upperFreq*, *PyCUDA=False*)
Filters data using fft -> zeroing out fft bins -> ifft

**Parameters**

**Signal** [ndarray] Signal to be filtered

**SampleFreq** [float] Sample frequency of signal

**lowerFreq** [float] Lower frequency of bandpass to allow through filter

**upperFreq** [float] Upper frequency of bandpass to allow through filter

**PyCUDA** [bool, optional] If True, uses PyCUDA to accelerate the FFT and IFFT via using your NVIDIA-GPU If False, performs FFT and IFFT with conventional scipy.fftpack

**Returns**

**FilteredData** [ndarray] Array containing the filtered data

optoanalysis.optoanalysis.**IIR_filter_design**(*CentralFreq*, *bandwidth*, *transitionWidth*, *SampleFreq*, *GainStop=40*, *GainPass=0.01*)
Function to calculate the coefficients of an IIR filter, IMPORTANT NOTE: make_butterworth_bandpass_b_a and make_butterworth_b_a can produce IIR filters with higher sample rates and are prefereable due to this.

**Parameters**

**CentralFreq** [float] Central frequency of the IIR filter to be designed

**bandwidth** [float] The width of the passband to be created about the central frequency

**transitionWidth** [float] The width of the transition band between the pass-band and stop-band

**SampleFreq** [float] The sample frequency (rate) of the data to be filtered

**GainStop** [float, optional] The dB of attenuation within the stopband (i.e. outside the passband)

**GainPass** [float, optional] The dB attenuation inside the passband (ideally close to 0 for a band-pass filter)

**Returns**

**b** [ndarray] coefficients multiplying the current and past inputs (feedforward coefficients)

**a** [ndarray] coefficients multiplying the past outputs (feedback coefficients)

**class** optoanalysis.optoanalysis.**ORGTableData**(*filename*)

Bases: object

Class for reading in general data from org-mode tables.

The table must be formatted as in the example below:

```
` | RunNo | ColumnName1 | ColumnName2 | |-------+-------------+-------------|
| 3 | 14 | 15e3 | `
```

In this case the run number would be 3 and the ColumnName2-value would be 15e3 (15000.0).

**get_value**(*ColumnName*, *RunNo*)

Retreives the value of the collumn named ColumnName associated with a particular run number.

**Parameters**

**ColumnName** [string] The name of the desired org-mode table's collumn

**RunNo** [int] The run number for which to retreive the pressure value

**Returns**

**Value** [float] The value for the column's name and associated run number

optoanalysis.optoanalysis.**PSD_fitting_eqn**(*A*, *OmegaTrap*, *Gamma*, *omega*)

The value of the fitting equation: A / ((OmegaTrap**2 - omega**2)**2 + (omega * Gamma)**2) to be fit to the PSD

**Parameters**

**A** [float] Fitting constant A A = $\gamma$**2*$\Gamma$_0*(2*K_b*T_0)/(m) where:

$\gamma$ = conversionFactor $\Gamma$_0 = Damping factor due to environment $\pi$ = pi

**OmegaTrap** [float] The trapping frequency in the axis of interest (in angular frequency)

**Gamma** [float] The damping factor Gamma = $\Gamma$ = $\Gamma$_0 + $\delta\Gamma$ where:

$\Gamma$_0 = Damping factor due to environment $\delta\Gamma$ = extra damping due to feedback or other effects

**omega** [float] The angular frequency to calculate the value of the fitting equation at

**Returns**

**Value** [float] The value of the fitting equation

optoanalysis.optoanalysis.**PSD_fitting_eqn2**(*A*, *OmegaTrap*, *Gamma*, *omega*)

The value of the fitting equation: A / ((OmegaTrap**2 - omega**2)**2 + (omega * Gamma)**2) to be fit to the PSD

**Parameters**

**A** [float] Fitting constant A A = $\gamma$**2*(2*K_b*T_0)/(m) where:

$\gamma$ = conversionFactor $\Gamma$_0 = Damping factor due to environment $\pi$ = pi

**OmegaTrap** [float] The trapping frequency in the axis of interest (in angular frequency)

**Gamma** [float] The damping factor Gamma = Γ = Γ_0 + δΓ where:

Γ_0 = Damping factor due to environment δΓ = extra damping due to feedback or other effects

**omega** [float] The angular frequency to calculate the value of the fitting equation at

**Returns**

**Value** [float] The value of the fitting equation

optoanalysis.optoanalysis.**PSD_fitting_eqn_with_background**(*A*, *OmegaTrap*, *Gamma*, *FlatBackground*, *omega*)

The value of the fitting equation: A / ((OmegaTrap**2 - omega**2)**2 + (omega * Gamma)**2) + FlatBackground to be fit to the PSD

**Parameters**

**A** [float] Fitting constant A A = γ**2*Γ_0*(2*K_b*T_0)/(m) where:

γ = conversionFactor Γ_0 = Damping factor due to environment π = pi

**OmegaTrap** [float] The trapping frequency in the axis of interest (in angular frequency)

**Gamma** [float] The damping factor Gamma = Γ = Γ_0 + δΓ where:

Γ_0 = Damping factor due to environment δΓ = extra damping due to feedback or other effects

**FlatBackground** [float] Adds a constant offset to the peak to account for a flat noise background

**omega** [float] The angular frequency to calculate the value of the fitting equation at

**Returns**

**Value** [float] The value of the fitting equation

optoanalysis.optoanalysis.**animate**(*zdata*, *xdata*, *ydata*, *conversionFactorArray*, *timedata*, *BoxSize*, *timeSteps=100*, *filename='particle'*)

Animates the particle's motion given the z, x and y signal (in Volts) and the conversion factor (to convert between V and nm).

**Parameters**

**zdata** [ndarray] Array containing the z signal in volts with time.

**xdata** [ndarray] Array containing the x signal in volts with time.

**ydata** [ndarray] Array containing the y signal in volts with time.

**conversionFactorArray** [ndarray] Array of 3 values of conversion factors for z, x and y (in units of Volts/Metre)

**timedata** [ndarray] Array containing the time data in seconds.

**BoxSize** [float] The size of the box in which to animate the particle - in nm

**timeSteps** [int, optional] Number of time steps to animate

**filename** [string, optional] filename to create the mp4 under (<filename>.mp4)

optoanalysis.optoanalysis.**animate_2Dscatter**(*x, y, NumAnimatedPoints=50, NTrailPoints=20, xlabel='', ylabel='', xlims=None, ylims=None, filename='testAnim.mp4', bitrate=100000.0, dpi=500.0, fps=30, figsize=[6, 6]*)

Animates x and y - where x and y are 1d arrays of x and y positions and it plots x[i:i+NTrailPoints] and y[i:i+NTrailPoints] against each other and iterates through i.

optoanalysis.optoanalysis.**animate_2Dscatter_slices**(*x, y, NumAnimatedPoints=50, xlabel='', ylabel='', xlims=None, ylims=None, filename='testAnim.mp4', bitrate=100000.0, dpi=500.0, fps=30, figsize=[6, 6]*)

Animates x and y - where x and y are both 2d arrays of x and y positions and it plots x[i] against y[i] and iterates through i.

optoanalysis.optoanalysis.**arrange_plots_on_one_canvas**(*FigureAxTupleArray, title='', SubtitleArray=[], number_of_columns=2, show_fig=True*)

Arranges plots, given in an array of tuples consisting of fig and axs, onto a subplot-figure consisting of number_of_columns horizontal times the lenght of the passed (fig,axs)-array divided by number_of_columns vertical subplots

**Parameters**

**FigureAxTupleArray** [array-like] array of Tuples(fig, axs) outputted from the other plotting funtions inside optoanalysis

**title** [string, optional] string for the global title of the overall combined figure

**SubtitleArray** [array-like, optional] array of titles for each figure-set to be plotted, i.e. subplots

**number_of_columns** [int, optional] Number of columns in the subplot grid By default set to 2 columns

**show_fig** [bool, optional] If True runs plt.show() before returning figure if False it just returns the figure object. (the default is True, it shows the figure)

**Returns**

**fig** [matplotlib.figure.Figure object] The figure object created

optoanalysis.optoanalysis.**audiate**(*signal, AudioSampleFreq, filename*)

optoanalysis.optoanalysis.**butterworth_filter**(*Signal, SampleFreq, lowerFreq, upperFreq*)

Filters data using by constructing a 5th order butterworth IIR filter and using scipy.signal.filtfilt, which does phase correction after implementing the filter (as IIR filter apply a phase change)

**Parameters**

**Signal** [ndarray] Signal to be filtered

**SampleFreq** [float] Sample frequency of signal

**lowerFreq** [float] Lower frequency of bandpass to allow through filter

**upperFreq** [float] Upper frequency of bandpass to allow through filter

**Returns**

**FilteredData** [ndarray] Array containing the filtered data

optoanalysis.optoanalysis.**calc_PSD**(*Signal*, *SampleFreq*, *NPerSegment=1000000*, *window='hann'*)

Extracts the pulse spectral density (PSD) from the data.

**Parameters**

**Signal** [array-like] Array containing the signal to have the PSD calculated for

**SampleFreq** [float] Sample frequency of the signal array

**NPerSegment** [int, optional] Length of each segment used in scipy.welch default = 1000000

**window** [str or tuple or array_like, optional] Desired window to use. See get_window for a list of windows and required parameters. If window is array_like it will be used directly as the window and its length will be used for nperseg. default = "hann"

**Returns**

**freqs** [ndarray] Array containing the frequencies at which the PSD has been calculated

**PSD** [ndarray] Array containing the value of the PSD at the corresponding frequency value in V**2/Hz

optoanalysis.optoanalysis.**calc_RSquared**(*time*, *Signal*, *SampleFreq*, *CenterFreq*, *CutOffFreq*, *FractionOfSampleFreq*)

Calculates the R Squared signal from a given Signal using demodulated signals and butterworth filtering.

**Parameters**

**time** [array-like] Array containing the time data points of the Signal

**Signal** [array-like] Array containing the signal to have the RSquared calculated for

**SampleFreq** [float] Sampling frequncy of the Signal

**CenterFreq** [float] central frequency of your Signal (oscillator)

**CutOffFreq** [float] is the cut off frequncy to get rid of the 2*omega component, make this several times larger than your linewidth, in Hz

**FractionOfSampleFreq** [integer, optional] The fraction of the sample frequency to sub-sample the data by. This sometimes needs to be done because a filter with the appropriate frequency response may not be generated using the sample rate at which the data was taken. Increasing this number means the R Squared signal produced by this function will be sampled at a lower rate but a higher number means a higher chance that the filter produced will have a nice frequency response.

**Returns**

**RSquared** [ndarray] Array containing the value of the RSquared signal

optoanalysis.optoanalysis.**calc_acceleration**(*xdata*, *dt*)

Calculates the acceleration from the position

**Parameters**

**xdata** [ndarray] Position data

**dt** [float] time between measurements

**Returns**

**acceleration** [ndarray] values of acceleration from position 2 to N.

optoanalysis.optoanalysis.**calc_autocorrelation**(*Signal*, *FFT=False*, *PyCUDA=False*)

Calculates the autocorrelation from a given Signal via using

**Parameters**

**Signal** [array-like] Array containing the signal to have the autocorrelation calculated for

**FFT** [optional, bool] Uses FFT to accelerate autocorrelation calculation, but assumes certain certain periodicity on the signal to autocorrelate. Zero-padding is added to account for this periodicity assumption.

**PyCUDA** [bool, optional] If True, uses PyCUDA to accelerate the FFT and IFFT via using your NVIDIA-GPU If False, performs FFT and IFFT with conventional scipy.fftpack

**Returns**

**Autocorrelation** [ndarray] Array containing the value of the autocorrelation evaluated at the corresponding amount of shifted array-index.

optoanalysis.optoanalysis.**calc_fft_with_PyCUDA**(*Signal*)

Calculates the FFT of the passed signal by using the scikit-cuda libary which relies on PyCUDA

**Parameters**

**Signal** [ndarray] Signal to be transformed into Fourier space

**Returns**

**Signalfft** [ndarray] Array containing the signal's FFT

optoanalysis.optoanalysis.**calc_gamma_components**(*Data_ref*, *Data*)

Calculates the components of Gamma (Gamma0 and delta_Gamma), assuming that the Data_ref is uncooled data (ideally at 3mbar for best fitting). It uses the fact that A_prime=A/Gamma0 should be constant for a particular particle under changes in pressure and therefore uses the reference save to calculate A_prime (assuming the Gamma value found for the uncooled data is actually equal to Gamma0 since only collisions should be causing the damping. Therefore for the cooled data Gamma0 should equal A/A_prime and therefore we can extract Gamma0 and delta_Gamma.

A_prime = ConvFactor**2 * (2*k_B*T0/(pi*m))

**Parameters**

**Data_ref** [DataObject] Reference data set, assumed to be 300K

**Data** [DataObject] Data object to have the temperature calculated for

**Returns**

**Gamma0** [uncertainties.ufloat] Damping due to the environment

**delta_Gamma** [uncertainties.ufloat] Damping due to other effects (e.g. feedback cooling)

optoanalysis.optoanalysis.**calc_ifft_with_PyCUDA**(*Signalfft*)

Calculates the inverse-FFT of the passed FFT-signal by using the scikit-cuda libary which relies on PyCUDA

**Parameters**

**Signalfft** [ndarray] FFT-Signal to be transformed into Real space

**Returns**

**Signal** [ndarray] Array containing the ifft signal

optoanalysis.optoanalysis.**calc_mass_from_fit_and_conv_factor**(*A*, *Damping*, *ConvFactor*)

Calculates mass from the A parameter from fitting, the damping from fitting in angular units and the Conversion factor calculated from comparing the ratio of the z signal and first harmonic of z.

**Parameters**

**A** [float] A factor calculated from fitting

**Damping** [float] damping in radians/second calcualted from fitting

**ConvFactor** [float] conversion factor between volts and nms

**Returns**

**mass** [float] mass in kgs

optoanalysis.optoanalysis.**calc_mass_from_z0**(*z0*, *w0*)

Calculates the mass of the particle using the equipartition from the angular frequency of the z signal and the average amplitude of the z signal in nms.

**Parameters**

**z0** [float] Physical average amplitude of motion in nms

**w0** [float] Angular Frequency of z motion

**Returns**

**mass** [float] mass in kgs

optoanalysis.optoanalysis.**calc_mean_amp**(*signal*)

calculates the mean amplitude by calculating the RMS of the signal and then multiplying it by 2.

**Parameters**

**signal** [ndarray]

**array of floats containing an AC signal**

**Returns**

**mean_amplitude** [float] the mean amplitude of the signal

optoanalysis.optoanalysis.**calc_radius_from_mass**(*Mass*)

Given the mass of a particle calculates the radius, assuming a 1800 kg/m**3 density.

**Parameters**

**Mass** [float] mass in kgs

**Returns**

**Radius** [float] radius in ms

optoanalysis.optoanalysis.**calc_reduced_chi_squared**(*y_observed*, *y_model*, *observation_error*, *number_of_fitted_parameters*)

Calculates the reduced chi-squared, used to compare a model to observations. For example can be used to calculate how good a fit is by using fitted y values for y_model along with observed y values and error in those y values. Reduced chi-squared should be close to 1 for a good fit, lower than 1 suggests you are overestimating the measurement error (observation_error you entered is higher than the true error in the measurement). A value higher than 1 suggests either your model is a bad fit OR you are underestimating the error in the measurement (observation_error you entered is lower than the true error in the measurement). See https://en.wikipedia.org/wiki/Reduced_chi-squared_statistic for more detail.

**Parameters**

**y_observed** [ndarray] array of measured/observed values of some variable y which you are fitting to.

**y_model** [ndarray] array of y values predicted by your model/fit (predicted y values corresponding to y_observed)

**observation_error** [float] error in the measurements/observations of y

**number_of_fitted_parameters** [float] number of parameters in your model

**Returns**

**chi2_reduced** [float] reduced chi-squared parameter

optoanalysis.optoanalysis.**calc_temp**(*Data_ref*, *Data*)

Calculates the temperature of a data set relative to a reference. The reference is assumed to be at 300K.

**Parameters**

**Data_ref** [DataObject] Reference data set, assumed to be 300K

**Data** [DataObject] Data object to have the temperature calculated for

**Returns**

**T** [uncertainties.ufloat] The temperature of the data set

optoanalysis.optoanalysis.**calc_z0_and_conv_factor_from_ratio_of_harmonics**(*z*, *z2*, *NA=0.999*)

Calculates the Conversion Factor and physical amplitude of motion in nms by comparison of the ratio of the heights of the z signal and second harmonic of z.

**Parameters**

**z** [ndarray] array containing z signal in volts

**z2** [ndarray] array containing second harmonic of z signal in volts

**NA** [float] NA of mirror used in experiment

**Returns**

**z0** [float] Physical average amplitude of motion in nms

**ConvFactor** [float] Conversion Factor between volts and nms

optoanalysis.optoanalysis.**count_collisions**(*Collisions*)

Counts the number of unique collisions and gets the collision index.

**Parameters**

**Collisions** [array_like] Array of booleans, containing true if during a collision event, false otherwise.

**Returns**

**CollisionCount** [int] Number of unique collisions

**CollisionIndicies** [list] Indicies of collision occurance

optoanalysis.optoanalysis.**dynamical_potential**(*xdata*, *dt*, *order=3*)

Computes potential from spring function

**Parameters**

**xdata** [ndarray] Position data for a degree of freedom, at which to calculate potential

**dt** [float] time between measurements

**order** [int] order of polynomial to fit

**Returns**

**Potential** [ndarray] valued of potential at positions in xdata

optoanalysis.optoanalysis.**extract_parameters**(*Pressure*, *PressureErr*, *A*, *AErr*, *Gamma0*, *Gamma0Err*, *method='chang'*)

Calculates the radius, mass and conversion factor and thier uncertainties. For values to be correct data must have been taken with feedback off and at pressures of around 1mbar (this is because the equations assume harmonic motion and at lower pressures the uncooled particle experiences anharmonic motion (due to exploring furthur outside the middle of the trap). When cooled the value of Gamma (the damping) is a combination of the enviromental damping and feedback damping and so is not the correct value for use in this equation (as it requires the enviromental damping). Environmental damping can be predicted though as A=const*Gamma0. By fitting to 1mbar data one can find the value of the const and therefore Gamma0 = A/const

**Parameters**

**Pressure** [float] Pressure in mbar when the data was taken

**PressureErr** [float] Error in the Pressure as a decimal (e.g. 15% error is 0.15)

**A** [float] Fitting constant A A = $\gamma$**2*2*Γ_0*(K_b*T_0)/($\pi$*m) where: $\gamma$ = conversionFactor Γ_0 = Damping factor due to environment $\pi$ = pi

**AErr** [float] Error in Fitting constant A

**Gamma0** [float] The enviromental damping factor Gamma_0 = Γ_0

**Gamma0Err** [float] The error in the enviromental damping factor Gamma_0 = Γ_0

**Returns:**

**Params** [list] [radius, mass, conversionFactor] The extracted parameters

**ParamsError** [list] [radiusError, massError, conversionFactorError] The error in the extracted parameters

optoanalysis.optoanalysis.**extract_slices**(*z*, *freq*, *sample_freq*, *show_plot=False*)

Iterates through z trace and pulls out slices of length period_samples and assigns them a phase from -180 to 180. Each slice then becomes a column in the 2d array that is returned. Such that the row (the first index) refers to phase (i.e. dat[0] are all the samples at phase = -180) and the column refers to the oscillation number (i.e. dat[:, 0] is the first oscillation).

**Parameters**

**z** [ndarray] trace of z motion

**freq** [float] frequency of motion

**sample_freq** [float] sample frequency of the z array

**show_plot** [bool, optional (default=False)] if true plots and shows the phase plotted against the positon for each oscillation built on top of each other.

**Returns**

**phase** [ndarray] phase (in degrees) for each oscillation

**phase_slices** [ndarray] 2d numpy array containing slices as detailed above.

optoanalysis.optoanalysis.**find_collisions**(*Signal*, *tolerance=50*)

Finds collision events in the signal from the shift in phase of the signal.

**Parameters**

**Signal** [array_like] Array containing the values of the signal of interest containing a single frequency.

**tolerance** [float] Percentage tolerance, if the value of the FM Discriminator varies from the mean by this percentage it is counted as being during a collision event (or the aftermath of an event).

**Returns**

**Collisions** [ndarray] Array of booleans, true if during a collision event, false otherwise.

optoanalysis.optoanalysis.**fit_PSD**(*Data*, *bandwidth*, *TrapFreqGuess*, *AGuess=1000000000.0*, *GammaGuess=400*, *FlatBackground=None*, *MakeFig=True*, *show_fig=True*, *plot_initial=True*)

Fits theory PSD to Data. Assumes highest point of PSD is the trapping frequency.

**Parameters**

**Data** [DataObject] data object to be fitted

**bandwidth** [float] bandwidth around trapping frequency peak to fit the theory PSD to

**TrapFreqGuess** [float] The approximate trapping frequency to use initially as the centre of the peak

**AGuess** [float, optional] The initial value of the A parameter to use in fitting

**GammaGuess** [float, optional] The initial value of the Gamma parameter to use in fitting

**FlatBackground** [float, optional] If given a number the fitting function assumes a flat background to get more exact Area, which does not factor in noise. defaults to None, which fits a model with no flat background contribution, basically no offset

**MakeFig** [bool, optional] Whether to construct and return the figure object showing the fitting. defaults to True

**show_fig** [bool, optional] Whether to show the figure object when it has been created. defaults to True

**Returns**

**ParamsFit - Fitted parameters:** [A, TrappingFrequency, Gamma, FlatBackground(optional)]

**ParamsFitErr - Error in fitted parameters:** [AErr, TrappingFrequencyErr, GammaErr, FlatBackgroundErr(optional)]

**fig** [matplotlib.figure.Figure object] figure object containing the plot

**ax** [matplotlib.axes.Axes object]

**axes with the data plotted of the:**

- initial data
- initial fit
- final fit

optoanalysis.optoanalysis.**fit_RSquared_PSD**(*RSquared_PSD*, *RSquared_freqs*, *GammaGuess*, *AGuess*, *OffsetGuess*, *CutOffFreq*, *Fit_xlim*, *MakeFig=True*, *show_fig=True*)

Fits equation 3 of paper (DOI: 10.1103/PhysRevResearch.2.023349) plus additive Offset to the computed PSD of the R Squared signal and returns the parameters with errors.

**Parameters**

**RSquared_PSD** [array] array containing PSD of the R Squared signal to be fitted

**RSquared_freqs** [array] array containing the frequencies of each point the PSD of the RSquared signal was computed

**GammaGuess** [float] The approximate Big Gamma (in radians) to use initially

**AGuess** [float] Inital guess for multiplicative factor for R^2 which equals: 8*(S_F/(2m^2*Omega0^2))^2

**OffsetGuess** [float] Additive Offset to the fitting equation.

**CutOffFreq** [float] is the cut off frequency applied via a lowpass filter when calculating the R Squared signal

**Fit_xlim** [list of float, optional] limits the R Squared PSD signal used for the fit function i.e.: [lowerLimit, upperLimit]

**MakeFig** [bool, optional] Whether to construct and return the figure object showing the fitting. defaults to True

**show_fig** [bool, optional] Whether to show the figure object when it has been created. defaults to True

**Returns**

**ParamsFit - Fitted parameters:** [A, Gamma, Offset]

**ParamsFitErr - Error in fitted parameters:** [AErr, GammaErr, OffsetErr]

**fig** [matplotlib.figure.Figure object] figure object containing the plot

**ax** [matplotlib.axes.Axes object]

**axes with the data plotted of the:**

- initial data
- final fit

optoanalysis.optoanalysis.**fit_autocorrelation**(*autocorrelation*, *time*, *GammaGuess*, *TrapFreqGuess=None*, *method='energy'*, *MakeFig=True*, *show_fig=True*)

Fits exponential relaxation theory to data.

**Parameters**

**autocorrelation** [array] array containing autocorrelation to be fitted

**time** [array] array containing the time of each point the autocorrelation was evaluated

**GammaGuess** [float] The approximate Big Gamma (in radians) to use initially

**TrapFreqGuess** [float] The approximate trapping frequency to use initially in Hz.

**method** [string, optional] To choose which autocorrelation fit is needed. 'position' : equation 4.20 from Tongcang Li's 2013 thesis

(DOI: 10.1007/978-1-4614-6031-2)

**'energy'** [proper exponential energy correlation decay] (DOI: 10.1103/PhysRevE.94.062151)

**MakeFig** [bool, optional] Whether to construct and return the figure object showing the fitting. defaults to True

**show_fig** [bool, optional] Whether to show the figure object when it has been created. defaults to True

**Returns**

**ParamsFit - Fitted parameters:** 'variance'-method : [Gamma] 'position'-method : [Gamma, AngularTrappingFrequency]

**ParamsFitErr - Error in fitted parameters:** 'varaince'-method : [GammaErr] 'position'-method : [GammaErr, AngularTrappingFrequencyErr]

**fig** [matplotlib.figure.Figure object] figure object containing the plot

**ax** [matplotlib.axes.Axes object]

**axes with the data plotted of the:**

- initial data
- final fit

optoanalysis.optoanalysis.**fit_curvefit**(*p0*, *datax*, *datay*, *function*, ***kwargs*)

Fits the data to a function using scipy.optimise.curve_fit

**Parameters**

**p0** [array_like] initial parameters to use for fitting

**datax** [array_like] x data to use for fitting

**datay** [array_like] y data to use for fitting

**function** [function] funcion to be fit to the data

**kwargs** keyword arguments to be passed to scipy.optimise.curve_fit

**Returns**

**pfit_curvefit** [array] Optimal values for the parameters so that the sum of the squared residuals of ydata is minimized

**perr_curvefit** [array] One standard deviation errors in the optimal values for the parameters

optoanalysis.optoanalysis.**fit_data**(*freq_array*, *S_xx_array*, *AGuess*, *OmegaTrap*, *GammaGuess*, *freq_range=None*, *make_fig=True*, *show_fig=True*, ***kwargs*)

optoanalysis.optoanalysis.**fit_data_2**(*freq_array*, *S_xx_array*, *AGuess*, *OmegaTrap*, *GammaGuess*, *make_fig=True*, *show_fig=True*)

optoanalysis.optoanalysis.**fit_radius_from_potentials**(*z*, *SampleFreq*, *Damping*, *HistBins=100*, *show_fig=False*)

Fits the dynamical potential to the Steady State Potential by varying the Radius.

**z** [ndarray] Position data

**SampleFreq** [float] frequency at which the position data was sampled

**Damping** [float] value of damping (in radians/second)

**HistBins** [int] number of values at which to evaluate the steady state potential / perform the fitting to the dynamical potential

**Returns**

**Radius** [float] Radius of the nanoparticle

**RadiusError** [float] One Standard Deviation Error in the Radius from the Fit (doesn't take into account possible error in damping)

**fig** [matplotlib.figure.Figure object] figure showing fitted dynamical potential and stationary potential

**ax** [matplotlib.axes.Axes object] axes for above figure

optoanalysis.optoanalysis.**fit_to_ringdown**(*time*, *signal*, *time_start*, *time_stop*, *Gamma_guess*)

optoanalysis.optoanalysis.**fm_discriminator**(*Signal*)

Calculates the digital FM discriminator from a real-valued time signal.

**Parameters**

**Signal** [array-like] A real-valued time signal

**Returns**

**fmDiscriminator** [array-like] The digital FM discriminator of the argument signal

optoanalysis.optoanalysis.**get_ZXY_data**(*Data*, *zf*, *xf*, *yf*, *FractionOfSampleFreq=1*, *zwidth=10000*, *xwidth=5000*, *ywidth=5000*, *filterImplementation='filtfilt'*, *timeStart=None*, *timeEnd=None*, *NPerSegmentPSD=1000000*, *MakeFig=True*, *show_fig=True*)

Given a Data object and the frequencies of the z, x and y peaks (and some optional parameters for the created filters) this function extracts the individual z, x and y signals (in volts) by creating IIR filters and filtering the Data.

**Parameters**

**Data** [DataObject] DataObject containing the data for which you want to extract the z, x and y signals.

**zf** [float] The frequency of the z peak in the PSD

**xf** [float] The frequency of the x peak in the PSD

**yf** [float] The frequency of the y peak in the PSD

**FractionOfSampleFreq** [integer, optional] The fraction of the sample frequency to sub-sample the data by. This sometimes needs to be done because a filter with the appropriate frequency response may not be generated using the sample rate at which the data was taken. Increasing this number means the x, y and z signals produced by this function will be sampled at a lower rate but a higher number means a higher chance that the filter produced will have a nice frequency response.

**zwidth** [float, optional] The width of the pass-band of the IIR filter to be generated to filter Z.

**xwidth** [float, optional] The width of the pass-band of the IIR filter to be generated to filter X.

**ywidth** [float, optional] The width of the pass-band of the IIR filter to be generated to filter Y.

**filterImplementation** [string, optional] filtfilt or lfilter - use scipy.filtfilt or lfilter default: filtfilt

**timeStart** [float, optional] Starting time for filtering

**timeEnd** [float, optional] Ending time for filtering

**show_fig** [bool, optional] If True - plot unfiltered and filtered PSD for z, x and y. If False - don't plot anything

**Returns**

**zdata** [ndarray] Array containing the z signal in volts with time.

**xdata** [ndarray] Array containing the x signal in volts with time.

**ydata** [ndarray] Array containing the y signal in volts with time.

**timedata** [ndarray] Array containing the time data to go with the z, x, and y signal.

optoanalysis.optoanalysis.**get_ZXY_data_IFFT**(*Data*, *zf*, *xf*, *yf*, *zwidth=10000*, *xwidth=5000*, *ywidth=5000*, *timeStart=None*, *timeEnd=None*, *show_fig=True*)

Given a Data object and the frequencies of the z, x and y peaks (and some optional parameters for the created filters) this function extracts the individual z, x and y signals (in volts) by creating IIR filters and filtering the Data.

**Parameters**

**Data** [DataObject] DataObject containing the data for which you want to extract the z, x and y signals.

**zf** [float] The frequency of the z peak in the PSD

**xf** [float] The frequency of the x peak in the PSD

**yf** [float] The frequency of the y peak in the PSD

**zwidth** [float, optional] The width of the pass-band of the IIR filter to be generated to filter Z.

**xwidth** [float, optional] The width of the pass-band of the IIR filter to be generated to filter X.

**ywidth** [float, optional] The width of the pass-band of the IIR filter to be generated to filter Y.

**timeStart** [float, optional] Starting time for filtering

**timeEnd** [float, optional] Ending time for filtering

**show_fig** [bool, optional] If True - plot unfiltered and filtered PSD for z, x and y. If False - don't plot anything

**Returns**

**zdata** [ndarray] Array containing the z signal in volts with time.

**xdata** [ndarray] Array containing the x signal in volts with time.

**ydata** [ndarray] Array containing the y signal in volts with time.

**timedata** [ndarray] Array containing the time data to go with the z, x, and y signal.

optoanalysis.optoanalysis.**get_ZXY_freqs**(*Data*, *zfreq*, *xfreq*, *yfreq*, *bandwidth=5000*)

Determines the exact z, x and y peak frequencies from approximate frequencies by finding the highest peak in the PSD "close to" the approximate peak frequency. By "close to" I mean within the range: approxFreq - bandwidth/2 to approxFreq + bandwidth/2

**Parameters**

**Data** [DataObject] DataObject containing the data for which you want to determine the z, x and y frequencies.

**zfreq** [float] An approximate frequency for the z peak

**xfreq** [float] An approximate frequency for the z peak

**yfreq** [float] An approximate frequency for the z peak

**bandwidth** [float, optional] The bandwidth around the approximate peak to look for the actual peak. The default value is 5000

**Returns**

**trapfreqs** [list] List containing the trap frequencies in the following order (z, x, y)

optoanalysis.optoanalysis.**get_freq_response**(*a*, *b*, *show_fig=True*, *SampleFreq=6.283185307179586*, *NumOfFreqs=500*, *whole=False*)

This function takes an array of coefficients and finds the frequency response of the filter using scipy.signal.freqz. show_fig sets if the response should be plotted

**Parameters**

**b** [array_like] Coefficients multiplying the x values (inputs of the filter)

**a** [array_like] Coefficients multiplying the y values (outputs of the filter)

**show_fig** [bool, optional] Verbosity of function (i.e. whether to plot frequency and phase response or whether to just return the values.) Options (Default is 1): False - Do not plot anything, just return values True - Plot Frequency and Phase response and return values

**SampleFreq** [float, optional] Sample frequency (in Hz) to simulate (used to convert frequency range to normalised frequency range)

**NumOfFreqs** [int, optional] Number of frequencies to use to simulate the frequency and phase response of the filter. Default is 500.

**Whole** [bool, optional] Sets whether to plot the whole response (0 to sample freq) or just to plot 0 to Nyquist (SampleFreq/2): False - (default) plot 0 to Nyquist (SampleFreq/2) True - plot the whole response (0 to sample freq)

**Returns**

**freqList** [ndarray] Array containing the frequencies at which the gain is calculated

**GainArray** [ndarray] Array containing the gain in dB of the filter when simulated (20*log_10(A_out/A_in))

**PhaseDiffArray** [ndarray] Array containing the phase response of the filter - phase difference between the input signal and output signal at different frequencies

optoanalysis.optoanalysis.**get_time_slice**(*time*, *z*, *zdot=None*, *timeStart=None*, *timeEnd=None*)

Get slice of time, z and (if provided) zdot from timeStart to timeEnd.

**Parameters**

**time** [ndarray] array of time values

**z** [ndarray] array of z values

**zdot** [ndarray, optional] array of zdot (velocity) values.

**timeStart** [float, optional] time at which to start the slice. Defaults to beginnging of time trace

**timeEnd** [float, optional] time at which to end the slide. Defaults to end of time trace

**Returns**

**time_sliced** [ndarray] array of time values from timeStart to timeEnd

**z_sliced** [ndarray] array of z values from timeStart to timeEnd

**zdot_sliced** [ndarray] array of zdot values from timeStart to timeEnd. None if zdot not provided

optoanalysis.optoanalysis.**get_wigner**(*z*, *freq*, *sample_freq*, *histbins=200*, *show_plot=False*)

Calculates an approximation to the wigner quasi-probability distribution by splitting the z position array into slices of the length of one period of the motion. This slice is then associated with phase from -180 to 180 degrees. These slices are then histogramed in order to get a distribution of counts of where the particle is observed at each phase. The 2d array containing the counts varying with position and phase is then passed through the

inverse radon transformation using the Simultaneous Algebraic Reconstruction Technique approximation from the scikit-image package.

**Parameters**

**z** [ndarray] trace of z motion

**freq** [float] frequency of motion

**sample_freq** [float] sample frequency of the z array

**histbins** [int, optional (default=200)] number of bins to use in histogramming data for each phase

**show_plot** [bool, optional (default=False)] Whether or not to plot the phase distribution

**Returns**

**iradon_output** [ndarray] 2d array of size (histbins x histbins)

**bin_centres** [ndarray] positions of the bin centres

optoanalysis.optoanalysis.**histogram_phase**(*phase_slices*, *phase*, *histbins=200*, *show_plot=False*)

histograms the phase slices such as to build a histogram of the position distribution at each phase value.

**Parameters**

**phase_slices** [ndarray] 2d array containing slices from many oscillations at each phase

**phase** [ndarray] 1d array of phases corresponding to slices

**histbins** [int, optional (default=200)] number of bins to use in histogramming data

**show_plot** [bool, optional (default=False)] if true plots and shows the heatmap of the phase against the positon distribution

**Returns**

**counts_array** [ndarray] 2d array containing the number of counts varying with phase and position.

**bin_edges** [ndarray] positions of bin edges

optoanalysis.optoanalysis.**load_data**(*Filepath*, *ObjectType='data'*, *RelativeChannelNo=None*, *SampleFreq=None*, *NumberOfChannels=None*, *PointsToLoad=-1*, *calcPSD=True*, *NPerSegmentPSD=1000000*, *NormaliseByMonitorOutput=False*, *silent=False*)

**Parameters**

**Filepath** [string] filepath to the file containing the data used to initialise and create an instance of the DataObject class

**ObjectType** [string, optional] type to load the data as, takes the value 'default' if not specified. Options are: 'data' : optoanalysis.DataObject 'thermo' : optoanalysis.thermo.ThermoObject

**RelativeChannelNo** [int, optional] If loading a .bin file produced by the Saneae datalogger, used to specify the channel number If loading a .mat file produced by the picoscope using picolog, used to specifiy the channel ID as follows: 0 = Channel 'A', 1 = Channel 'B', 2 = Channel 'C' and 3 = Channel 'D' If loading a .bin file saved using custom code to interface with the Picoscope used to specify the channel number to load in conjunction with the NumberOfChannels parameter, if left None with .bin files it will assume that the file to load only contains one channel. If loading a .dat file produced by the labview NI5122 daq card,

used to specifiy the channel number if two channels where saved, if left None with .dat files it will assume that the file to load only contains one channel. If NormaliseByMonitorOutput is True then RelativeChannelNo specifies the monitor channel for loading a .dat file produced by the labview NI5122 daq card.

**SampleFreq** [float, optional] Manual selection of sample frequency for loading labview NI5122 daq files and .mat and .bin files recorded using the Picoscope

**NumberOfChannels** [int, optional] Total number of channels present in a .bin file recorded using a Picoscope.

**PointsToLoad** [int, optional] Number of first points to read. -1 means all points (i.e., the complete file) WORKS WITH NI5122 AND PICOSCOPE .BIN DATA SO FAR ONLY!!!

**calcPSD** [bool, optional] Whether to calculate the PSD upon loading the file, can take some time off the loading and reduce memory usage if frequency space info is not required

**NPerSegmentPSD** [int, optional] NPerSegment to pass to scipy.signal.welch to calculate the PSD

**NormaliseByMonitorOutput** [bool, optional] If True the particle signal trace will be divided by the monitor output, which is specified by the channel number set in the RelativeChannelNo parameter. WORKS WITH NI5122 DATA SO FAR ONLY!!!

**Returns**

**Data** [DataObject] An instance of the DataObject class contaning the data that you requested to be loaded.

optoanalysis.optoanalysis.**make_butterworth_b_a**(*lowcut*, *highcut*, *SampleFreq*, *order=5*, *btype='band'*)

Generates the b and a coefficients for a butterworth IIR filter.

**Parameters**

**lowcut** [float] frequency of lower bandpass limit

**highcut** [float] frequency of higher bandpass limit

**SampleFreq** [float] Sample frequency of filter

**order** [int, optional] order of IIR filter. Is 5 by default

**btype** [string, optional] type of filter to make e.g. (band, low, high)

**Returns**

**b** [ndarray] coefficients multiplying the current and past inputs (feedforward coefficients)

**a** [ndarray] coefficients multiplying the past outputs (feedback coefficients)

optoanalysis.optoanalysis.**make_butterworth_bandpass_b_a**(*CenterFreq*, *bandwidth*, *SampleFreq*, *order=5*, *btype='band'*)

Generates the b and a coefficients for a butterworth bandpass IIR filter.

**Parameters**

**CenterFreq** [float] central frequency of bandpass

**bandwidth** [float] width of the bandpass from centre to edge

**SampleFreq** [float] Sample frequency of filter

**order** [int, optional] order of IIR filter. Is 5 by default

**btype** [string, optional] type of filter to make e.g. (band, low, high)

**Returns**

**b** [ndarray] coefficients multiplying the current and past inputs (feedforward coefficients)

**a** [ndarray] coefficients multiplying the past outputs (feedback coefficients)

optoanalysis.optoanalysis.**make_dynamical_potential_func**(*kBT_Gamma*, *density*, *SpringPotnlFunc*)

Creates the function that calculates the potential given the position (in volts) and the radius of the particle.

**Parameters**

**kBT_Gamma** [float] Value of kB*T/Gamma

**density** [float] density of the nanoparticle

**SpringPotnlFunc** [function] Function which takes the value of position (in volts) and returns the spring potential

**Returns**

**PotentialFunc** [function] function that calculates the potential given the position (in volts) and the radius of the particle.

optoanalysis.optoanalysis.**moving_average**(*array*, *n=3*)

Calculates the moving average of an array.

**Parameters**

**array** [array] The array to have the moving average taken of

**n** [int] The number of points of moving average to take

**Returns**

**MovingAverageArray** [array] The n-point moving average of the input array

optoanalysis.optoanalysis.**multi_load_data**(*Channel*, *RunNos*, *RepeatNos*, *directoryPath='.'*, *calcPSD=True*, *NPerSegmentPSD=1000000*)

Lets you load multiple datasets at once assuming they have a filename which contains a pattern of the form: CH<ChannelNo>_RUN00...<RunNo>_REPEAT00...<RepeatNo>

**Parameters**

**Channel** [int] The channel you want to load

**RunNos** [sequence] Sequence of run numbers you want to load

**RepeatNos** [sequence] Sequence of repeat numbers you want to load

**directoryPath** [string, optional] The path to the directory housing the data The default is the current directory

**Returns**

**Data** [list] A list containing the DataObjects that were loaded.

optoanalysis.optoanalysis.**multi_load_data_custom**(*Channel*, *TraceTitle*, *RunNos*, *directoryPath='.'*, *calcPSD=True*, *NPerSegmentPSD=1000000*)

Lets you load multiple datasets named with the LeCroy's custom naming scheme at once.

**Parameters**

**Channel** [int] The channel you want to load

**TraceTitle** [string] The custom trace title of the files.

**RunNos** [sequence] Sequence of run numbers you want to load

**RepeatNos** [sequence] Sequence of repeat numbers you want to load

**directoryPath** [string, optional] The path to the directory housing the data The default is the current directory

**Returns**

**Data** [list] A list containing the DataObjects that were loaded.

optoanalysis.optoanalysis.**multi_plot_3d_dist**(*ZXYData*, *N=1000*, *AxisOffset=0*, *Angle=-40*, *LowLim=None*, *HighLim=None*, *ColorArray=None*, *alphaLevel=0.3*, *show_fig=True*)

Plots serveral Z, X and Y datasets as a 3d scatter plot with heatmaps of each axis pair in each dataset.

**Parameters**

**ZXYData** [ndarray] Array of arrays containing Z, X, Y data e.g. [[Z1, X1, Y1], [Z2, X2, Y2]]

**N** [optional, int] Number of time points to plot (Defaults to 1000)

**AxisOffset** [optional, double] Offset to add to each axis from the data - used to get a better view of the heat maps (Defaults to 0)

**LowLim** [optional, double] Lower limit of x, y and z axis

**HighLim** [optional, double] Upper limit of x, y and z axis

**show_fig** [optional, bool] Whether to show the produced figure before returning

**Returns**

**fig** [matplotlib.figure.Figure object] The figure object created

**ax** [matplotlib.axes.Axes object] The subplot object created

optoanalysis.optoanalysis.**multi_plot_PSD**(*DataArray, xlim=[0, 500], units='kHz', LabelArray=[], ColorArray=[], alphaArray=[], show_fig=True*)

plot the pulse spectral density for multiple data sets on the same axes.

**Parameters**

**DataArray** [array-like] array of DataObject instances for which to plot the PSDs

**xlim** [array-like, optional] 2 element array specifying the lower and upper x limit for which to plot the Power Spectral Density

**units** [string] units to use for the x axis

**LabelArray** [array-like, optional] array of labels for each data-set to be plotted

**ColorArray** [array-like, optional] array of colors for each data-set to be plotted

**show_fig** [bool, optional] If True runs plt.show() before returning figure if False it just returns the figure object. (the default is True, it shows the figure)

**Returns**

**fig** [matplotlib.figure.Figure object] The figure object created

**ax** [matplotlib.axes.Axes object] The axes object created

optoanalysis.optoanalysis.**multi_plot_time**(*DataArray*, *SubSampleN=1*, *units='s'*, *xlim=None*, *ylim=None*, *LabelArray=[]*, *show_fig=True*)

plot the time trace for multiple data sets on the same axes.

**Parameters**

**DataArray** [array-like] array of DataObject instances for which to plot the PSDs

**SubSampleN** [int, optional] Number of intervals between points to remove (to sub-sample data so that you effectively have lower sample rate to make plotting easier and quicker.

**xlim** [array-like, optional] 2 element array specifying the lower and upper x limit for which to plot the time signal

**LabelArray** [array-like, optional] array of labels for each data-set to be plotted

**show_fig** [bool, optional] If True runs plt.show() before returning figure if False it just returns the figure object. (the default is True, it shows the figure)

**Returns**

**fig** [matplotlib.figure.Figure object] The figure object created

**ax** [matplotlib.axes.Axes object] The axes object created

optoanalysis.optoanalysis.**multi_subplots_time**(*DataArray*, *SubSampleN=1*, *units='s'*, *xlim=None*, *ylim=None*, *LabelArray=[]*, *show_fig=True*)

plot the time trace on multiple axes

**Parameters**

**DataArray** [array-like] array of DataObject instances for which to plot the PSDs

**SubSampleN** [int, optional] Number of intervals between points to remove (to sub-sample data so that you effectively have lower sample rate to make plotting easier and quicker.

**xlim** [array-like, optional] 2 element array specifying the lower and upper x limit for which to plot the time signal

**LabelArray** [array-like, optional] array of labels for each data-set to be plotted

**show_fig** [bool, optional] If True runs plt.show() before returning figure if False it just returns the figure object. (the default is True, it shows the figure)

**Returns**

**fig** [matplotlib.figure.Figure object] The figure object created

**axs** [list of matplotlib.axes.Axes objects] The list of axes object created

optoanalysis.optoanalysis.**parse_orgtable**(*lines*)

Parse an org-table (input as a list of strings split by newline) into a Pandas data frame.

**Parameters**

**lines** [string] an org-table input as a list of strings split by newline

**Returns**

**dataframe** [pandas.DataFrame] A data frame containing the org-table's data

optoanalysis.optoanalysis.**plot_3d_dist**(*Z*, *X*, *Y*, *N=1000*, *AxisOffset=0*, *Angle=-40*, *LowLim=None*, *HighLim=None*, *show_fig=True*)

Plots Z, X and Y as a 3d scatter plot with heatmaps of each axis pair.

**Parameters**

**Z** [ndarray] Array of Z positions with time

**X** [ndarray] Array of X positions with time

**Y** [ndarray] Array of Y positions with time

**N** [optional, int] Number of time points to plot (Defaults to 1000)

**AxisOffset** [optional, double] Offset to add to each axis from the data - used to get a better view of the heat maps (Defaults to 0)

**LowLim** [optional, double] Lower limit of x, y and z axis

**HighLim** [optional, double] Upper limit of x, y and z axis

**show_fig** [optional, bool] Whether to show the produced figure before returning

**Returns**

**fig** [matplotlib.figure.Figure object] The figure object created

**ax** [matplotlib.axes.Axes object] The subplot object created

optoanalysis.optoanalysis.**plot_wigner2d**(*iradon_output*, *bin_centres*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *figsize=(6, 6)*)

Plots the wigner space representation as a 2D heatmap.

**Parameters**

**iradon_output** [ndarray] 2d array of size (histbins x histbins)

**bin_centres** [ndarray] positions of the bin centres

**cmap** [matplotlib.cm.cmap, optional (default=cm.cubehelix_r)] color map to use for Wigner

**figsize** [tuple, optional (default=(6, 6))] tuple defining size of figure created

**Returns**

**fig** [matplotlib.figure.Figure object] figure showing the wigner function

**ax** [matplotlib.axes.Axes object] axes containing the object

optoanalysis.optoanalysis.**plot_wigner3d**(*iradon_output*, *bin_centres*, *bin_centre_units=''*, *cmap=<matplotlib.colors.LinearSegmentedColormap object>*, *view=(10, -45)*, *figsize=(10, 10)*)

Plots the wigner space representation as a 3D surface plot.

**Parameters**

**iradon_output** [ndarray] 2d array of size (histbins x histbins)

**bin_centres** [ndarray] positions of the bin centres

**bin_centre_units** [string, optional (default="")] Units in which the bin_centres are given

**cmap** [matplotlib.cm.cmap, optional (default=cm.cubehelix_r)] color map to use for Wigner

**view** [tuple, optional (default=(10, -45))] view angle for 3d wigner plot

**figsize** [tuple, optional (default=(10, 10))] tuple defining size of figure created

**Returns**

**fig** [matplotlib.figure.Figure object] figure showing the wigner function

**ax** [matplotlib.axes.Axes object] axes containing the object

optoanalysis.optoanalysis.**search_data_custom**(*Channel*, *TraceTitle*, *RunNos*, *directoryPath='.'*)

Lets you create a list with full file paths of the files named with the LeCroy's custom naming scheme.

**Parameters**

**Channel** [int] The channel you want to load

**TraceTitle** [string] The custom trace title of the files.

**RunNos** [sequence] Sequence of run numbers you want to load

**RepeatNos** [sequence] Sequence of repeat numbers you want to load

**directoryPath** [string, optional] The path to the directory housing the data The default is the current directory

**Returns**

**Paths** [list] A list containing the full file paths of the files you were looking for.

optoanalysis.optoanalysis.**search_data_std**(*Channel*, *RunNos*, *RepeatNos*, *directoryPath='.'*)

Lets you find multiple datasets at once assuming they have a filename which contains a pattern of the form: CH<ChannelNo>_RUN00...<RunNo>_REPEAT00...<RepeatNo>

**Parameters**

**Channel** [int] The channel you want to load

**RunNos** [sequence] Sequence of run numbers you want to load

**RepeatNos** [sequence] Sequence of repeat numbers you want to load

**directoryPath** [string, optional] The path to the directory housing the data The default is the current directory

**Returns**

**Data_filepaths** [list] A list containing the filepaths to the matching files

optoanalysis.optoanalysis.**steady_state_potential**(*xdata*, *HistBins=100*)

Calculates the steady state potential. Used in fit_radius_from_potentials.

**Parameters**

**xdata** [ndarray] Position data for a degree of freedom

**HistBins** [int] Number of bins to use for histogram of xdata. Number of position points at which the potential is calculated.

**Returns**

**position** [ndarray] positions at which potential has been calculated

**potential** [ndarray] value of potential at the positions above

optoanalysis.optoanalysis.**take_closest**(*myList*, *myNumber*)

Assumes myList is sorted. Returns closest value to myNumber. If two numbers are equally close, return the smallest number.

**Parameters**

**myList** [array] The list in which to find the closest value to myNumber

**myNumber** [float] The number to find the closest to in MyList

**Returns**

**closestValue** [float] The number closest to myNumber in myList

optoanalysis.optoanalysis.**unit_conversion**(*array*, *unit_prefix*, *current_prefix=''*)

Converts an array or value to of a certain unit scale to another unit scale.

Accepted units are: E - exa - 1e18 P - peta - 1e15 T - tera - 1e12 G - giga - 1e9 M - mega - 1e6 k - kilo - 1e3 m - milli - 1e-3 u - micro - 1e-6 n - nano - 1e-9 p - pico - 1e-12 f - femto - 1e-15 a - atto - 1e-18

**Parameters**

**array** [ndarray] Array to be converted

**unit_prefix** [string] desired unit (metric) prefix (e.g. nm would be n, ms would be m)

**current_prefix** [optional, string] current prefix of units of data (assumed to be in SI units by default (e.g. m or s)

**Returns**

**converted_array** [ndarray] Array multiplied such as to be in the units specified

CHAPTER 2

# optoanalysis.thermo package

## 2.1 optoanalysis.thermo.thermo module

**class** optoanalysis.thermo.thermo.**ThermoObject**(*filepath*, *RelativeChannelNo=None*, *SampleFreq=None*, *NumberOfChannels=None*, *PointsToLoad=-1*, *calcPSD=True*, *NPerSegmentPSD=1000000*, *NormaliseByMonitorOutput=False*)

Bases: *optoanalysis.optoanalysis.DataObject*

Creates an object containing some data and all it's properties for thermodynamics analysis.

**Attributes**

**SampleFreq** [float]

The sample frequency used in generating the data.

**time** [ndarray] Contains the time data in seconds

**voltage** [ndarray] Contains the voltage data in Volts - with noise and clean signals all added together

**SampleFreq** [sample frequency used to sample the data (when it was] taken by the oscilloscope)

**freqs** [ndarray] Contains the frequencies corresponding to the PSD (Pulse Spectral Density)

**PSD** [ndarray] Contains the values for the PSD (Pulse Spectral Density) as calculated at each frequency contained in freqs

**calc_hamiltonian = <MagicMock name='mock()' id='140300074248336'>**

**calc_mean_and_variance_of_variances**(*NumberOfOscillations*)

Calculates the mean and variance of a set of varainces. This set is obtained by splitting the timetrace into chunks of points with a length of NumberOfOscillations oscillations.

**Parameters**

**NumberOfOscillations** [int] The number of oscillations each chunk of the timetrace used to calculate the variance should contain.

**Returns**

**Mean** [float]

**Variance** [float]

**calc_phase_space_density = <MagicMock name='mock()' id='140300074248336'>**

**extract_thermodynamic_quantities = <MagicMock name='mock()' id='140300074248336'>**

# CHAPTER 3

# optoanalysis.LeCroy package

## 3.1 optoanalysis.LeCroy.LeCroy module

**class** optoanalysis.LeCroy.LeCroy.**HDO6104**(*address='152.78.194.16'*)

Bases: object

Class for communicating with the Teledyne LeCroy Oscilloscope.

**data**(*channel=1*)

Reads the raw input from the scope and interprets it returning the header information, time, voltage and raw integers read with the ADC.

**Parameters**

**channel** [int] channel number of read

**Returns**

**WAVEDESC** [dict] dictionary containing some properties of the time trace and oscilloscope settings extracted from the header file.

**x** [ndarray] The array of time values recorded by the oscilloscope

**y** [ndarray] The array of voltage values recorded by the oscilloscope

**integers** [ndarray] The array of raw integers recorded from the ADC and stored in the binary file

**opc**()

Asks the oscilloscope if it is done processing data.

**Returns**

**IsDoneProcessing** [bool] returns False if oscilloscope is still busy, True is oscilloscope is done processing last commands.

**raw**(*channel=1*)

Reads the raw input from the oscilloscope.

**Parameters**

**channel** [int] channel number of read

**Returns**

**rawData** [bytes] raw binary data read from the oscilloscope

**waitOPC**()

Waits for a response from the oscilloscope indicating that processing is complete and it is ready to receive more commands. Function sleeps until the oscilloscope is ready.

optoanalysis.LeCroy.LeCroy.**InterpretWaveform**(*raw*, *integersOnly=False*, *headersOnly=False*, *noTimeArray=False*)

Take the raw binary from a file saved from the LeCroy, read from a file using the 2 lines: with open(filename, "rb") as file: raw = file.read() And extracts various properties of the saved time trace.

**Parameters**

**raw** [bytes] Bytes object containing the binary contents of the saved raw/trc file

**integersOnly** [bool, optional] If True, only returns the unprocessed integers (read from the ADC) rather than the signal in volts. Defaults to False.

**headersOnly** [bool, optional] If True, only returns the file header. Defaults to False.

**noTimeArray** [bool, optional] If true returns timeStart, timeStop and timeStep and doesn't create the time array

**Returns**

**WAVEDESC** [dict] dictionary containing some properties of the time trace and oscilloscope settings extracted from the header file.

**x** [ndarray / tuple] The array of time values recorded by the oscilloscope or, if noTimeArray is True, returns a tuplef of (timeStart, timeStop, timeStep)

**y** [ndarray] The array of voltage values recorded by the oscilloscope

**integers** [ndarray] The array of raw integers recorded from the ADC and stored in the binary file

**MissingData** [bool] bool stating if any data was missing

CHAPTER 4

# optoanalysis.Saleae package

## 4.1 optoanalysis.Saleae.Saleae module

optoanalysis.Saleae.Saleae.**get_chunks**(*Array*, *Chunksize*)
Generator that yields chunks of size ChunkSize

optoanalysis.Saleae.Saleae.**interpret_waveform**(*fileContent*, *RelativeChannelNo*)
Extracts the data for just 1 channel and computes the corresponding time array (in seconds) starting from 0.

Important Note: RelativeChannelNo is NOT the channel number on the Saleae data logger it is the relative number of the channel that was saved. E.g. if you save channels 3, 7 and 10, the corresponding RelativeChannelNos would be 0, 1 and 2.

**Parameters**

**fileContent** [bytes] bytes object containing the data from a .bin file exported from the saleae data logger.

**RelativeChannelNo** [int] The relative order/position of the channel number in the saved binary file. See Important Note above!

**Returns**

**time** [ndarray] A generated time array corresponding to the data list

**Data** [list] The data from the relative channel requested

**SampleTime** [float] The time between samples (in seconds)

optoanalysis.Saleae.Saleae.**read_data_from_bin_file**(*fileName*)
Loads the binary data stored in the a binary file and extracts the data for each channel that was saved, along with the sample rate and length of the data array.

**Parameters**

**fileContent** [bytes] bytes object containing the data from a .bin file exported from the saleae data logger.

**Returns**

**ChannelData** [list] List containing a list which contains the data from each channel

**LenOf1Channel** [int] The length of the data in each channel

**NumOfChannels** [int] The number of channels saved

**SampleTime** [float] The time between samples (in seconds)

**SampleRate** [float] The sample rate (in Hz)

optoanalysis.Saleae.Saleae.**read_data_from_bytes**(*fileContent*)

Takes the binary data stored in the binary string provided and extracts the data for each channel that was saved, along with the sample rate and length of the data array.

**Parameters**

**fileContent** [bytes] bytes object containing the data from a .bin file exported from the saleae data logger.

**Returns**

**ChannelData** [list] List containing a list which contains the data from each channel

**LenOf1Channel** [int] The length of the data in each channel

**NumOfChannels** [int] The number of channels saved

**SampleTime** [float] The time between samples (in seconds)

**SampleRate** [float] The sample rate (in Hz)

---

- genindex
- modindex
- search

# Python Module Index

## o

# Index

## F



## G



## H



## I



## L



## M

## O



## P



## R



## S



## T



## U



## W