pwspy Documentation

Release 0.2.15.dev9+g4422d8d

Nick Anthony

Mar 03, 2022

CONTENTS

PWSpy is a Python library for working with Partial Wave Spectroscopic Microscopy data. It provides a concise and simple interface for loading and analyzing experimental data. Support for modern as well as legacy file formats of PWS data is provided.

With PWSpy, it is trivial to skip to skip the basics and get to the heart of extracting meaningful results from your experimental data. Basic operations such as normalization, hardware compensation, and calibration are handled with the call of a single method. Additionally, the library provides a means for conveniently loading and storing auxiliary data such as ROIs, notes, and analysis outputs.

Utility functionality for generating visualizations, automatic colocalization, calculation of thin-film reflectance based on fresnel equations, parsing metadata from the graph-based acquisition engine, and more are provided in the utility subpackage.

CHAPTER

ONE

USAGE

Almost any usage of PWSpy will start with loading of experimental data. The simplest way to do this is with the *pwspy.dataTypes.Acquisition* class. An instance of *Acquisition* provides properties (*pws*, *dynamics*, *fluorescence*) which provide references to colocalized measurements which may be part of a single acquisition.

```
import pwspy.dataTypes as pwsdt

acq = pwsdt.Acquisition(pathToData)

pwsMetadata = acq.pws

if acq.dynamics is not None:
    dynamicsMetadata = acq.dynamics

if acq.fluorescence is not None:
    listOfFluorMetadata = acq.fluorescence
```

An acquisition also provides access to information that applies to all measurements such as ROIs and automated imaging metadata.

```
roiInfos: List[Tuple[str, int, Roi.FileFormat]] = acq.getRois()
for roiInfo in roiInfos:
    roiFile = acq.loadRoi(*roiInfo)
```

Each sub-measurement metadata object of an *Acquisition* provides access to information specific to that measurement such as analysis results and raw data.

```
from pwspy.analysis.pws import PWSAnalysisResults
from pwspy.analysis.dynamics import DynAnalysisResults

listOfAnalysisNames = acq.pws.getAnalyses()
analysisName = listOfAnalysisNames[0]

pAnalysisResults: PWSAnalysisResults = acq.pws.loadAnalysis(analysisName)
dAnalysisResults: DynAnalysisResults = acq.dynamics.loadAnalysis(analysisName)
roi: pwsdt.Roi = acq.loadRoi(roiName, roiNumber).getRoi()

print(f"Average nuclear Sigma is: {pAnalysisResults.rms[roi.mask].mean()}")
print(f"Average nuclear Sigma_t^2 is: {dAnalysisResults.rms_t_squared[roi.mask].mean()}")
```

There is much more functionality in this library, please see the examples and API documentation.

CHAPTER

TWO

API

analysis	Contains all code used for the analysis of data acquired with the PWS system.
dataTypes	Custom datatypes that are commonly used in the analysis of PWS related data.
utility	Useful subpackages ranging many different topics

2.1 pwspy.analysis

Contains all code used for the analysis of data acquired with the PWS system.

2.1.1 Submodules

compilation	Classes used during the "compilation" step of analysis.
pws	Classes used in the analysis of PWS data.
warnings	Functions which check data during analysis for abnormality.
dynamics	Classes used in the analysis of Dynamics data.

pwspy.analysis.compilation

Classes used during the “compilation” step of analysis. This is when the data saved during analysis is combined with ROIs to generate a table of values such as the average RMS, reflectance, diffusion coefficient, etc.

PWS

PWSCompilerSettings([reflectance, rms, ...])	These settings determine which values should be processed during compilation
PWSRoiCompilationResults(cellIdTag, ...)	
PWSRoiCompiler(settings)	

pwspy.analysis.compilation.PWSCompilerSettings

class pwspy.analysis.compilation.**PWSCompilerSettings**(*reflectance=False*, *rms=False*, *polynomialRms=False*, *autoCorrelationSlope=False*, *rSquared=False*, *ld=False*, *opd=False*, *meanSigmaRatio=False*)

Bases: pwspy.analysis.compilation._abstract.AbstractCompilerSettings

These settings determine which values should be processed during compilation

pwspy.analysis.compilation.PWSRoiCompilationResults

class pwspy.analysis.compilation.**PWSRoiCompilationResults**(*cellIdTag*, *analysisName*, *reflectance*, *rms*, *polynomialRms*, *autoCorrelationSlope*, *rSquared*, *ld*, *opd*, *opdIndex*, *varRatio*)

Bases: pwspy.analysis.compilation._abstract.AbstractRoiCompilationResults

pwspy.analysis.compilation.PWSRoiCompiler

class pwspy.analysis.compilation.**PWSRoiCompiler**(*settings*)

Bases: pwspy.analysis.compilation._abstract.AbstractRoiCompiler

run(*results*, *roi*)

Combine information from analysis results and an ROI to produce values averaged over the ROI.

Parameters

- **results** (PWSAnalysisResults) – The analysis results to compile.
- **roi** (Roi) – The ROI to be used to segment out a section of the results.

Return type t_.Tuple[PWSRoiCompilationResults, t_.List[warnings.AnalysisWarning]]

Dynamics

DynamicsCompilerSettings([meanReflectance, ...])	These settings determine how a Dynamics acquisition should be compiled.
DynamicsRoiCompilationResults(cellIdTag, ...)	
DynamicsRoiCompiler(settings)	

pwspy.analysis.compilation.DynamicsCompilerSettings

class pwspy.analysis.compilation.**DynamicsCompilerSettings**(*meanReflectance=False*, *rms_t_squared=False*, *diffusion=False*)

Bases: pwspy.analysis.compilation._abstract.AbstractCompilerSettings

These settings determine how a Dynamics acquisition should be compiled.

pwspy.analysis.compilation.DynamicsRoiCompilationResults

class pwspy.analysis.compilation.**DynamicsRoiCompilationResults**(*cellIdTag*, *analysisName*, *reflectance*, *rms_t_squared*, *diffusion*)

Bases: pwspy.analysis.compilation._abstract.AbstractRoiCompilationResults

pwspy.analysis.compilation.DynamicsRoiCompiler

class pwspy.analysis.compilation.**DynamicsRoiCompiler**(*settings*)

Bases: pwspy.analysis.compilation._abstract.AbstractRoiCompiler

run(*results*, *roi*)

Combine information from analysis results and an ROI to produce values averaged over the ROI.

Parameters

- **results** (*DynamicsAnalysisResults*) – The analysis results to compile.
- **roi** (*Roi*) – The ROI to be used to segment out a section of the results.

Return type Tuple[*DynamicsRoiCompilationResults*, List[AnalysisWarning]]

Generic

GenericCompilerSettings([roiArea])	These settings determine which values should be processed during compilation
GenericRoiCompilationResults(roiFile, roiArea)	Results for compilation that don't pertain to any specific analysis.
GenericRoiCompiler(settings)	

pwspy.analysis.compilation.GenericCompilerSettings

class pwspy.analysis.compilation.**GenericCompilerSettings**(*roiArea=False*)

Bases: pwspy.analysis.compilation._abstract.AbstractCompilerSettings

These settings determine which values should be processed during compilation

pwspy.analysis.compilation.GenericRoiCompilationResults

class pwspy.analysis.compilation.**GenericRoiCompilationResults**(*roiFile*, *roiArea*)
Bases: pwspy.analysis.compilation._abstract.AbstractRoiCompilationResults

Results for compilation that don't pertain to any specific analysis.

pwspy.analysis.compilation.GenericRoiCompiler

class pwspy.analysis.compilation.**GenericRoiCompiler**(*settings*)
Bases: object

pwspy.analysis.pws

Classes used in the analysis of PWS data.

Classes

PWSAnalysisSettings(filterOrder, ...)	These settings determine the behavior of the PWSAnalysis class.
PWSAnalysisResults([file, variablesDict, ...])	A representation of analysis results.
PWSAnalysis(settings, extraReflectance, ref)	The standard PWS analysis routine.

pwspy.analysis.pws.PWSAnalysisSettings

class pwspy.analysis.pws.**PWSAnalysisSettings**(*filterOrder*, *filterCutoff*, *polynomialOrder*, *extraReflectanceId*, *referenceMaterial*, *wavelengthStart*, *wavelengthStop*, *skipAdvanced*, *autoCorrStopIndex*, *autoCorrMinSub*, *numericalAperture*, *relativeUnits*, *cameraCorrection*, *waveNumberCutoff*)
Bases: pwspy.analysis._abstract.AbstractAnalysisSettings

These settings determine the behavior of the PWSAnalysis class.

filterOrder
The *order* of the buttersworth filter used for lowpass filtering.

Type int

filterCutoff
The cutoff frequency of the buttersworth filter used for lowpass filtering. Frequency unit is 1/wavelength . Set to *None* to skip lowpass filtering.

Type float

polynomialOrder
The order of the polynomial which will be fit to the reflectance and then subtracted before calculating the analysis results.

Type int

extraReflectanceId
The *idtag* of the extra reflection used for correction. Set to *None* if extra reflectance calibration is being

skipped.

Type str

referenceMaterial
The material that was being imaged in the reference acquisition

Type *Material*

wavelengthStart
The acquisition spectra will be truncated at this wavelength before analysis. Set to *None* to bypass this step

Type int

wavelengthStop
The acquisition spectra will be truncated after this wavelength before analysis. Set to *None* to bypass this step

Type int

skipAdvanced
If *True* then skip analysis of the OPD and autocorrelation.

Type bool

autoCorrStopIndex
The autocorrelation slope will be calculated up to this number of elements. More elements is theoretically better but it severely limited by SNR.

Type int

autoCorrMinSub
If *True* then subtract the minimum of the ACF from ACF. This prevents numerical issues but doesn't actually make any sense.

Type bool

numericalAperture
The numerical aperture that the acquisition was imaged at.

Type float

relativeUnits
relativeUnits: If *True* then all calculation are performed such that the reflectance is 1 if it matches the reference. If *False* then we use the theoretical reflectance of the reference (based on NA and reference material) to normalize our results to the actual physical reflectance of the sample (about 0.4% for water)

Type bool

cameraCorrection
An object describing the dark counts and non-linearity of the camera used. If the data supplied to the PWSAnalysis class has already been corrected then this setting will not be used. Setting this to *None* will result in the camera correcting being automatically determined based on the image files' metadata.

Type Optional[*pwspy.dataTypes._other.CameraCorrection*]

waveNumberCutoff
A cutoff frequency for filtering the signal after converting from wavelength to wavenumber. In units of microns (opd). Note: To convert from depth to opd divide by 2 (because the light makes a round trip) and divide by the RI of the media (nucleus)

Type float

asDict()

Return type dict

Returns A dictionary with setting names as the keys and the values of the settings as the values.

classmethod fromJson(*filePath*, *name*)
Create a new instance of this class from a json text file.

Parameters

- **filePath** (str) – The path to the folder containing the JSON file to be loaded.
- **name** (str) – The name that the analysis was saved as.

Return type AbstractAnalysisSettings

Returns A new instance of an analysis settings class.

classmethod fromJsonString(*string*)
Use *_fromDict* to load a new instance of the *cls* from a json string.

Parameters **string** (str) – A JSON formatted string to load the object from.

Return type AbstractAnalysisSettings

Returns A new instance of analysis settings class.

toJson(*filePath*, *name*)
Save this object to a json text file.

Parameters

- **filePath** (str) – The path to the folder to contain the new JSON file.
- **name** (str) – The name to save the analysis as.

toJsonString()
Use *_asDict* to convert an instance of this class to a json string.

Return type str

Returns A JSON formatted string.

pwspy.analysis.pws.PWSAnalysisResults

class pwspy.analysis.pws.**PWSAnalysisResults**(*file=None*, *variablesDict=None*, *analysisName=None*)
Bases: pwspy.analysis._abstract.AbstractHDFAnalysisResults

A representation of analysis results. Items are loaded from disk using lazy-loading strategy and are then cached in memory.

static FieldDecorator(*func*)
Decorate functions in subclasses that access their fields from the HDF file with this decorator. It will: 1: Make it so the data is load from disk on the first access and stored in memory for every further access. 2: Report an understandable error if the field isn't found in the HDF file. 3: Make the accessors work even if the the object isn't associated with an HDF file.

classmethod create(*settings*, *reflectance*, *meanReflectance*, *rms*, *polynomialRms*, *autoCorrelationSlope*, *rSquared*, *ld*, *imCubeIdTag*, *referenceIdTag*, *extraReflectionTag*)
Used to create results from existing variables. These results can then be saved to file.

Returns A new instance of analysis results.

static fields()

Returns A sequence of string names of the datafields that the analysis results contains.

`static fileName2Name(fileName)`

Parameters **fileName** (str) – The filename that the HDF file was saved as.

Return type str

Returns The analysis name.

`classmethod load(directory, name)`
Load an analyis results object from an HDF5 file located in *directory*.

Parameters

- **directory** (str) – The path to the folder containing the file.
- **name** (str) – The name of the analysis.

Return type AbstractHDFAnalysisResults

Returns A new instance of analysis results loaded from file.

`static name2FileName(name)`

Parameters **name** (str) – An analysis name.

Return type str

Returns The corresponding file name for the hdf5 file.

`releaseMemory()`
The cached properties continue to stay in RAM until they are deleted, this method deletes all cached data to release the memory.

`toHDF(directory, name, overwrite=False, compression=None)`
Save the AnalysisResults object to an HDF file in *directory*. The name of the file will be determined by *name*. If you want to know what the full file name will be you can use this class's *name2FileName* method.

Parameters

- **directory** (str) – The path to the folder to save the file in.
- **name** (str) – The name of the analysis. This determines the file name.
- **overwrite** (bool) – If *True* then any existing file of the same name will be replaced.
- **compression** (Optional[str]) – The value of this argument will be passed to h5py.create_dataset for numpy arrays. See h5py documentation for available options.

`autoCorrelationSlope`
A 2D array giving the slope of the ACF of the spectra at each position in the image.

`extraReflectionTag`
The *idtag* of the extra reflectance correction used.

`imCubeIdTag`
The idtag of the acquisition that was analyzed.

`ld`
A 2D array giving Ld. A parameter derived from RMS and the ACF slope.

`meanReflectance`
A 2D array giving the reflectance of the image averaged over the full spectra.

moduleVersion
The version of PWSpy code that this file was saved with.

opd
The 3D array of values, *opdIndex*: The sequence of OPD values associated with each 2D slice along the 3rd axis of the *opd* data.

Type A tuple containing

Type *opd*

polynomialRms
A 2D array giving the variance of the polynomial fit that was subtracted from the reflectance before calculating RMS.

rSquared
A 2D array giving the r^2 coefficient of determination for the linear fit to the logarithm of the ACF. This basically tells us how confident to be in the *autoCorrelationSlope*.

referenceIdTag
The idtag of the acquisition that was used as a reference for normalization.

reflectance
The KCube containing the 3D reflectance data after all corrections and analysis.

rms
A 2D array giving the spectral variance at each posiiton in the image.

settings
The settings used for the analysis

time
The time that the analysis was performed.

pwspy.analysis.pws.PWSAnalysis

class pwspy.analysis.pws.**PWSAnalysis**(*settings*, *extraReflectance*, *ref*)
Bases: pwspy.analysis._abstract.AbstractAnalysis

The standard PWS analysis routine. Initialize and then *run* for as many different PwsCubes as you want. For a given set of settings and reference you only need to instantiate one instance of this class. You can then perform *run* on as many data cubes as you want.

Parameters

- **settings** (*PWSAnalysisSettings*) – The settings used for the analysis
- **extraReflectance** (Union[*ERMetaData*, *ExtraReflectanceCube*, *ExtraReflectionCube*, None]) – An object used to correct for stray reflectance present in the imaging system. This can be of type: None: No correction will be performed. ERMetaData (Recommended): The metadata object referring to a calibration file for extra reflectance. It will be processed in conjunction with the reference immage to produce an ExtraReflectionCube representing the stray reflectance in units of camera counts/ms. ExtraReflectanceCube: Effectively identical to supplying an ERMataData object. ExtraReflectionCube: An object representing the stray reflection in units of counts/ms. It is up to the user to make sure that the data is scaled appropriately to match the data being analyzed.
- **ref** (*PwsCube*) – The reference acquisition used for analysis.

copySharedDataToSharedMemory()
When running the *run* method in parallel memory for the object used must be copied to each new process. We can avoid that and save a lot of Ram by moving data that is shared between processes to shared memory. If you don't want to implement this then just override it and raise NotImplementedError

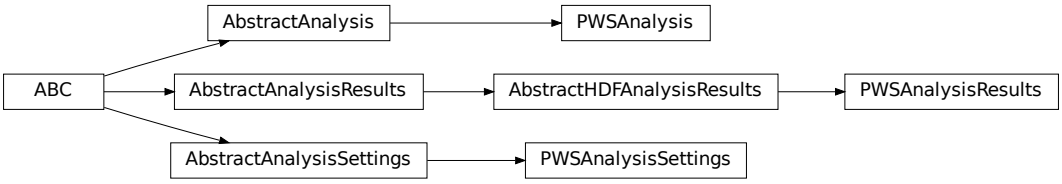
run(*cube*)
Given an data cube to analyze this function returns an instanse of AnalysisResults. In the PWSAnalysisApp this function is run in parallel by the AnalysisManager.

Parameters **cube** (*PwsCube*) – A data cube to be analyzed using the settings provided in the constructor of this class.

Return type Tuple[*PWSAnalysisResults*, List[AnalysisWarning]]

Returns A new instance of analysis results.

Inheritance



pwspy.analysis.warnings

Functions which check data during analysis for abnormality. If abnormal conditions are found then an AnalysisWarning is produced. The application can then display or record these warnings. This aspect of the program has not really been fully implemented.

pwspy.analysis.dynamics

Classes used in the analysis of Dynamics data.

Classes

DynamicsAnalysisSettings(extraReflectanceId, ...)	These settings determine the behavior of the *Dynamics-Analysis* class.
DynamicsAnalysisResults([file, ...])	
DynamicsAnalysis(settings, extraReflectance, ref)	This class performs the analysis of RMS_t_squared and D (diffusion).

pwspy.analysis.dynamics.DynamicsAnalysisSettings

class pwspy.analysis.dynamics.**DynamicsAnalysisSettings**(*extraReflectanceId*, *referenceMaterial*, *numericalAperture*, *relativeUnits*, *cameraCorrection*, *diffusionRegressionLength=3*)

Bases: pwspy.analysis._abstract.AbstractAnalysisSettings

These settings determine the behavior of the *DynamicsAnalysis* class.

Parameters

- **extraReflectanceId** (Optional[str]) – The unique *IDTag* of the extraReflectance calibration that was used on this analysis.
- **referenceMaterial** (*Material*) – The material that was imaged in the reference image of this analysis. Found as an in pwspy.moduleConst.Material. The theoretically predicted reflectance of the reference image is used in the extraReflectance correction.
- **numericalAperture** (float) – The illumination NA of the system. This is used for two purposes. First, we want to make sure that the NA of our data matches the NA of our extra reflectance correction cube. Second, the theoretically predicted reflectance of our reference is based not only on what our refereMaterial is but also the NA since reflectance is angle dependent.
- **relativeUnits** (bool) – If *True* then all calculation are performed such that the reflectance is 1 if it matches the reference. If *False* then we use the theoretical reflectance of the reference (based on NA and reference material) to normalize our results to the actual physical reflectance of the sample (about 0.4% for water)
- **cameraCorrection** (Optional[*CameraCorrection*]) – An object describing the dark counts and non-linearity of the camera used. If the data supplied to the DynamicsAnalysis class has already been corrected then this setting will not be used. Setting this to *None* will result in the camera correcting being automatically determined based on the image files' metadata.
- **diffusionRegressionLength** (int) – The original matlab scripts for analysis of dynamics data determined the slope of the log(ACF) by looking only at the first two indices, (log(ACF)[1]-log(ACF)[0])/dt. This results in very noisy results. However as you at higher index value of the log(ACF) the noise becomes much worse. A middle ground is to perform linear regression on the first 4 indices to determine the slope. You can adjust that number here.

asDict()

Return type dict

Returns A dictionary with setting names as the keys and the values of the settings as the values.

classmethod **fromJson**(*filePath*, *name*)

Create a new instance of this class from a json text file.

Parameters

- **filePath** (str) – The path to the folder containing the JSON file to be loaded.
- **name** (str) – The name that the analysis was saved as.

Return type AbstractAnalysisSettings

Returns A new instance of an analysis settings class.

classmethod fromJsonString(*string*)
Use *_fromDict* to load a new instance of the *cls* from a json string.

Parameters **string** (str) – A JSON formatted string to load the object from.

Return type AbstractAnalysisSettings

Returns A new instance of analysis settings class.

toJson(*filePath*, *name*)
Save this object to a json text file.

Parameters

- **filePath** (str) – The path to the folder to contain the new JSON file.
- **name** (str) – The name to save the analysis as.

toJsonString()
Use *_asDict* to convert an instance of this class to a json string.

Return type str

Returns A JSON formatted string.

pwspy.analysis.dynamics.DynamicsAnalysisResults

class pwspy.analysis.dynamics.**DynamicsAnalysisResults**(*file=None*, *variablesDict=None*, *analysisName=None*)
Bases: pwspy.analysis._abstract.AbstractHDFAnalysisResults

static FieldDecorator(*func*)
Decorate functions in subclasses that access their fields from the HDF file with this decorator. It will: 1: Make it so the data is load from disk on the first access and stored in memory for every further access. 2: Report an understandable error if the field isn't found in the HDF file. 3: Make the accessors work even if the the object isn't associated with an HDF file.

classmethod create(*settings*, *meanReflectance*, *rms_t_squared*, *reflectance*, *diffusion*, *imCubeIdTag*, *referenceIdTag*, *extraReflectionIdTag*)
Used to create results from existing variables. These results can then be saved to file.

Returns A new instance of analysis results.

static fields()

Returns A sequence of string names of the datafields that the analysis results contains.

static fileName2Name(*fileName*)

Parameters **fileName** (str) – The filename that the HDF file was saved as.

Return type str

Returns The analysis name.

classmethod load(*directory*, *name*)
Load an analyis results object from an HDF5 file located in *directory*.

Parameters

- **directory** (str) – The path to the folder containing the file.

- **name** (str) – The name of the analysis.

Return type AbstractHDFAnalysisResults

Returns A new instance of analysis results loaded from file.

static name2FileName(*name*)

Parameters **name** (str) – An analysis name.

Return type str

Returns The corresponding file name for the hdf5 file.

toHDF(*directory*, *name*, *overwrite=False*, *compression=None*)

Save the AnalysisResults object to an HDF file in *directory*. The name of the file will be determined by *name*. If you want to know what the full file name will be you can use this class's *name2FileName* method.

Parameters

- **directory** (str) – The path to the folder to save the file in.
- **name** (str) – The name of the analysis. This determines the file name.
- **overwrite** (bool) – If *True* then any existing file of the same name will be replaced.
- **compression** (Optional[str]) – The value of this argument will be passed to h5py.create_dataset for numpy arrays. See h5py documentation for available options.

diffusion

A 2D array indicating the diffusion at each position in the image.

extraReflectionIdTag

The idtag of the extra reflection correction that was used.

imCubeIdTag

The idtag of the dynamics cube that was analyzed.

meanReflectance

A 2D array giving the reflectance of the image averaged over the full spectra.

moduleVersion

The version of PWSpy code that this file was saved with.

referenceIdTag

The idtag of the dynamics cube that was used as a reference for normalization.

reflectance

A dynamics cube containing the 3D reflectance array after all corrections and analysis.

rms_t_squared

A 2D array giving the spectral variance at each position in the image.

settings

The settings used to generate these results.

time

The time that the analysis was performed.

pwspy.analysis.dynamics.DynamicsAnalysis

class pwspy.analysis.dynamics.**DynamicsAnalysis**(*settings*, *extraReflectance*, *ref*)

Bases: pwspy.analysis._abstract.AbstractAnalysis

This class performs the analysis of RMS_t_squared and D (diffusion). It is based on a set of MATLAB scripts written by Scott Gladstein. The original scripts can be found in the *_oldMatlab* subpackage.

References

"Multimodal interferometric imaging of nanoscale structure and macromolecular motion uncovers UV induced cellular paroxysm"

Parameters

- **settings** (*DynamicsAnalysisSettings*) – The settings use for the analysis
- **extraReflectance** (Union[*ERMetaData*, *ExtraReflectanceCube*, None]) – the metadata object referring to a calibration file for extra reflectance. You can optionally proide the ExtraReflectanceCube rather than just the metadata object referring to it.
- **ref** (*DynCube*) – A reference acquisition to use for normalization.

copySharedDataToSharedMemory()

When running the *run* method in parallel memory for the object used must be copied to each new process. We can avoid that and save a lot of Ram by moving data that is shared between processes to shared memory. If you don't want to implement this then just override it and raise NotImplementedError

run(*cube*)

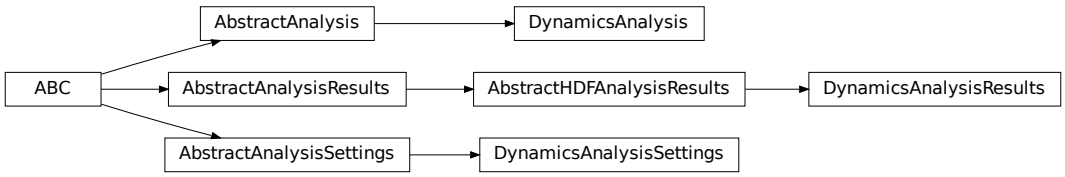
Given an data cube to analyze this function returns an instanse of AnalysisResults. In the PWSAnalysisApp this function is run in parallel by the AnalysisManager.

Parameters **cube** (*DynCube*) – A data cube to be analyzed using the settings provided in the constructor of this class.

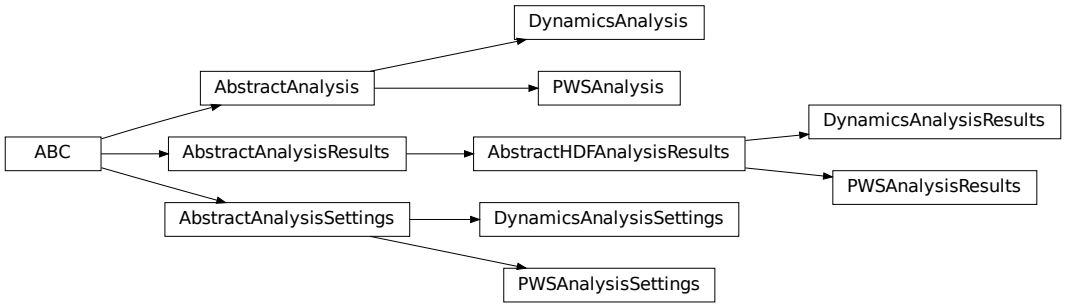
Return type Tuple[*DynamicsAnalysisResults*, List[AnalysisWarning]]

Returns A new instance of analysis results.

Inheritance



2.1.2 Inheritance



2.2 pwspy.dataTypes

Custom datatypes that are commonly used in the analysis of PWS related data.

2.2.1 Metadata Classes

These classes provide handling of information about an acquisition without requiring that the full data be loaded into RAM. These can be used to get information about the equipment used, the date the acquisition was taken, the location of the files, the presence of ROIs or analyses, etc.

PwsMetaData	A class that represents the metadata of a PWS acquisition.
DynMetaData	A class that represents the metadata of a Dynamics acquisition.
ERMetaData	A class representing the extra information related to an ExtraReflectanceCube file.
FluorMetaData	Metadata for a fluorescence image.

pwspy.dataTypes.PwsMetaData

class pwspy.dataTypes.**PwsMetaData**(*metadata*, *filePath=None*, *fileFormat=None*, *acquisitionDirectory=None*)

Bases: pwspy.dataTypes._metadata.MetaDataBase, pwspy.dataTypes._metadata.AnalysisManager

A class that represents the metadata of a PWS acquisition.

Parameters **metadata** (dict) – The dictionary containing the metadata.

class **FileFormats**(*value*)

Bases: enum.Enum

An enumeration.

static decodeHdfMetadata(*d*)
Attempt to extract a dictionary of metadata from an HDF5 dataset.

Parameters **d** (Dataset) – The *h5py.Dataset* to load from.

Return type dict

Returns A dictionary containing the metadata

encodeHdfMetadata(*d*)
Save this metadata object as a json string in an HDF5 dataset.

Parameters **d** (Dataset) – The *h5py.Dataset* to save the metadata to.

Return type Dataset

classmethod fromNano(*directory*, *lock=None*, *acquisitionDirectory=None*)
Attempt to load from NanoCytomic .mat file format

Parameters **directory** (str) – The file path to load the metadata from.

Return type *PwsMetaData*

Returns A new instance of *PwsMetaData* loaded from file

classmethod fromOldPWS(*directory*, *lock=None*, *acquisitionDirectory=None*)
Attempt to load from the old .mat file format.

Parameters **directory** – The file path to load the metadata from.

Return type *PwsMetaData*

Returns A new instance of *PwsMetaData* loaded from file

classmethod fromTiff(*directory*, *lock=None*, *acquisitionDirectory=None*)
Attempt to load from the standard TIFF file format.

Parameters **directory** – The file path to load the metadata from.

Return type *PwsMetaData*

Returns A new instance of *PwsMetaData* loaded from file

getAnalyses()

Return type List[str]

Returns A list of the names of analyses that were found.

classmethod getAnalysesAtPath(*path*)

Parameters **path** (str) – The path to search for analysis files.

Return type List[str]

Returns A list of the names of analyses that were found.

static getAnalysisResultsClass()

Return type t_.Type[AbstractHDFAnalysisResults]

Returns The class that is used to contain analysis results for this acquisition type.

getThumbnail()

Return type ndarray

Returns An image for quick viewing of the acquisition. No numerical significance.

loadAnalysis(*name*)

Parameters **name** (*str*) – The name of the analysis to load.

Return type AbstractHDFAnalysisResults

Returns A new instance of an AnalysisResults object.

classmethod loadAny(*directory*, *lock=None*, *acquisitionDirectory=None*)
Attempt to load from any file format.

Parameters **directory** – The file path to load the metadata from.

Return type *PwsMetaData*

Returns A new instance of *PwsMetaData* loaded from file

metadataToJson(*directory*)
Save the metadata to a JSON file.

Parameters **directory** – The folder path to save the new file to.

removeAnalysis(*name*)

Parameters **name** (str) – The name of the analysis to be deleted

saveAnalysis(*analysis*, *name*, *overwrite=False*)

Parameters

- **analysis** (*AbstractHDFAnalysisResults*) – An AnalysisResults object to be saved.
- **name** (*str*) – The name to save the analysis as
- **overwrite** (*bool*) – If *True* then any existing file of the same name will be replaced. If *False* an exception will be raised.

toDataClass(*lock=None*)
Convert the metadata class to a class that loads the data

Parameters **lock** (Optional[Lock]) – A *Lock* object used to synchronize IO in multithreaded and multiprocessing applications.

Return type *PwsCube*

property binning: int
The binning setting used by the camera. This is needed in order to properly correct dark counts. This is generally extracted from metadata saved by Micromanager

Return type int

property exposure: float
The exposure time of the camera expressed in milliseconds.

Return type float

idTag

property pixelSizeUm: float
The pixelSize expressed in microns. This represents the length of each square pixel in object space. Binning has already been accounted for here. This is generally extracted from metadata saved my MicroManager

Return type float

property systemName: str
The name of the system this was acquired on. The name is set in the *PWS Acquisition Plugin* for Micromanager.

Return type str

property time: str
The date and time that the acquisition was taken.

Return type str

pwspy.dataTypes.DynMetaData

class pwspy.dataTypes.**DynMetaData**(*metadata*, *filePath=None*, *fileFormat=None*, *acquisitionDirectory=None*)
Bases: pwspy.dataTypes._metadata.MetaDataBase, pwspy.dataTypes._metadata.AnalysisManager

A class that represents the metadata of a Dynamics acquisition.

class FileFormats(*value*)
Bases: enum.Enum

An enumerator identifying the types of file formats that this class can be loaded from.

static decodeHdfMetadata(*d*)
Attempt to extract a dictionary of metadata from an HDF5 dataset.

Parameters **d** (Dataset) – The *h5py.Dataset* to load from.

Return type dict

Returns A dictionary containing the metadata

encodeHdfMetadata(*d*)
Save this metadata object as a json string in an HDF5 dataset.

Parameters **d** (Dataset) – The *h5py.Dataset* to save the metadata to.

Return type Dataset

classmethod fromOldPWS(*directory*, *lock=None*, *acquisitionDirectory=None*)
Loads old dynamics cubes which were saved the same as old pws cubes. a raw binary file with some metadata saved in random .mat files. Does not support automatic detection of binning, pixel size, camera dark counts, system name.

Parameters **directory** (str) – The path to the folder containing the data files load the metadata from.

Return type *DynMetaData*

Returns A new instance of *DynMetaData*.

classmethod fromTiff(*directory*, *lock=None*, *acquisitionDirectory=None*)

Parameters **directory** – The path to the folder containing the data files load the metadata from.

Return type *DynMetaData*

Returns A new instance of *DynMetaData* loaded from file.

`getAnalyses()`

Return type List[str]

Returns A list of the names of analyses that were found.

`classmethod getAnalysesAtPath(path)`

Parameters **path** (str) – The path to search for analysis files.

Return type List[str]

Returns A list of the names of analyses that were found.

`static getAnalysisResultsClass()`

Return type t_.Type[AbstractHDFAnalysisResults]

Returns The class that is used to contain analysis results for this acquisition type.

`getThumbnail()`
Return the image used for quick viewing of the acquisition. Has no numeric significance.

Return type ndarray

`loadAnalysis(name)`

Parameters **name** (*str*) – The name of the analysis to load.

Return type AbstractHDFAnalysisResults

Returns A new instance of an AnalysisResults object.

`removeAnalysis(name)`

Parameters **name** (str) – The name of the analysis to be deleted

`saveAnalysis(analysis, name, overwrite=False)`

Parameters

- **analysis** (*AbstractHDFAnalysisResults*) – An AnalysisResults object to be saved.
- **name** (*str*) – The name to save the analysis as
- **overwrite** (*bool*) – If *True* then any existing file of the same name will be replaced. If *False* an exception will be raised.

`toDataClass(lock=None)`

Parameters **lock** (*mp.Lock*) – A multiprocessing *lock* that can prevent help us synchronize demands on the hard drive when loading many files in parallel. Probably not needed.

Returns The data object associated with this metadata object.

Return type pwsdtmd.DynCube

property binning: int
The binning setting used by the camera. This is needed in order to properly correct dark counts. This is generally extracted from metadata saved by Micromanager

Return type int

property exposure: float
The exposure time of the camera expressed in milliseconds.

Return type float

property idTag: str
Returns: str: A unique string identifying this acquisition.

Return type str

property pixelSizeUm: float
The pixelSize expressed in microns. This represents the length of each square pixel in object space. Binning has already been accounted for here. This is generally extracted from metadata saved my MicroManager

Return type float

property systemName: str
The name of the system this was acquired on. The name is set in the *PWS Acquisition Plugin* for Micromanager.

Return type str

property time: str
The date and time that the acquisition was taken.

Return type str

property times: Tuple[float, ...]
A sequence indicatin the time associated with each 2D slice of the 3rd axis of the *data* array

Return type Tuple[float, ...]

property wavelength: int
The wavelength that this acquisition was acquired at.

Return type int

pwspy.dataTypes.ERMetaData

class pwspy.dataTypes.**ERMetaData**(*inheritedMetadata*, *numericalAperture*, *filePath=None*)
Bases: object

A class representing the extra information related to an ExtraReflectanceCube file. This can be useful as a object to keep track of a ExtraReflectanceCube without having to have the data from the file loaded into memory.

Parameters

- **inheritedMetadata** (dict) – The metadata dictionary will often just be inherited information from one of the *PwsCubes* that was used to create this ER Cube. While this data can be useful it should be taken with a grain of salt. E.G. the metadata will contain an *exposure* field. In reality this ER Cube will have been created from pwsdtd.PwsCubes at a variety of exposures.
- **numericalAperture** (float) – The numerical aperture that the PwsCubes used to generate this Extra reflection cube were imaged at.
- **filePath** (Optional[str]) – The path to the file that this object is stored in.

classmethod dirName2Directory(*directory*, *name*)
This is the inverse of *directory2dirName*

Return type str

classmethod directory2dirName(*path*)

Parameters **path** (str) – The path to the file that stores an *ExtraReflectanceCube* object.

Returns directory: The directory path, name: The name that the file was saved as.

Return type A tuple containing

classmethod fromHdfDataset(*d*, *filePath=None*)

Parameters **d** (Dataset) – The *h5py.Dataset* to load the object from.

Return type *ERMetaData*

Returns A new instance of *ERMetaData* object

classmethod fromHdfFile(*directory*, *name*)

Parameters

- **directory** (str) – The directory the file is saved in.
- **name** (str) – The name the object was saved as.

Return type *ERMetaData*

Returns A new instance of *ERMetaData* object

toHdfDataset(*g*)

Parameters **g** (Group) – The *h5py.Group* to save the new dataset into.

Return type Group

classmethod validPath(*path*)

Parameters **path** (str) – The file path to the file to search for valid ExtraReflectance files.

Returns validPath: True if the path is valid, directory: The directory the file is in, name: The name that the object was saved as.

Return type A tuple containing

property idTag: str
A unique tag to identify this acquisition by.

Return type str

property numericalAperture: float
The numerical aperture that this cube was imaged at.

Return type float

property systemName: str
The name of the system that this image was acquired on.

Return type str

pwspy.dataTypes.FluorMetaData

class pwspy.dataTypes.**FluorMetaData**(*md*, *filePath=None*, *acquisitionDirectory=None*)
Bases: pwspy.dataTypes._metadata.MetaDataBase

Metadata for a fluorescence image.

Parameters **md** (dict) – A dictionary containing the metadata

static **decodeHdfMetadata**(*d*)
Attempt to extract a dictionary of metadata from an HDF5 dataset.

Parameters **d** (Dataset) – The *h5py.Dataset* to load from.

Return type dict

Returns A dictionary containing the metadata

encodeHdfMetadata(*d*)
Save this metadata object as a json string in an HDF5 dataset.

Parameters **d** (Dataset) – The *h5py.Dataset* to save the metadata to.

Return type Dataset

classmethod **fromTiff**(*directory*, *acquisitionDirectory*)
Load from a TIFF file.

Parameters **directory** (str) – The path to the folder to load from.

Return type *FluorMetaData*

getThumbnail()

Return type ndarray

Returns An image for quick viewing of the acquisition. No numerical significance.

classmethod **isValidPath**(*directory*)

Parameters **directory** (str) – The path to search for valid files.

Returns True if a valid file was found.

toDataClass(*lock=None*)
Convert the metadata class to a class that loads the data

Parameters **lock** (Optional[Lock]) – A *Lock* object used to synchronize IO in multithreaded and multiprocessing applications.

Return type *FluorescenceImage*

property **binning: int**
The binning setting used by the camera. This is needed in order to properly correct dark counts. This is generally extracted from metadata saved by Micromanager

Return type int

property **exposure: float**
The exposure time of the camera expressed in milliseconds.

Return type float

property idTag
A string that uniquely identifies this data.

property pixelSizeUm: float
The pixelSize expressed in microns. This represents the length of each square pixel in object space. Binning has already been accounted for here. This is generally extracted from metadata saved my MicroManager

Return type float

property systemName: str
The name of the system this was acquired on. The name is set in the *PWS Acquisition Plugin* for Micromanager.

Return type str

property time: str
The date and time that the acquisition was taken.

Return type str

2.2.2 Data Classes

These classes are used to actuallly load and manipulate acquisition data. The all have a corresponding metadata class.

PwsCube	A class representing a single PWS acquisition.
DynCube	A class representing a single acquisition of PWS Dynamics.
KCube	A class representing an PwsCube after being transformed from being described in terms of wavelength to wavenumber (k-space).
ExtraReflectanceCube	This class represents a 3D data cube of the extra reflectance in a PWS system.
ExtraReflectionCube	This class is meant to be constructed from an ExtraReflectanceCube along with additional reference measurement information.
ICBase	A class to handle the data operations common to PWS related *image cubes*.
ICRawBase	This class represents data cubes which are not derived from other data cubes.

pwspy.dataTypes.PwsCube

class pwspy.dataTypes.**PwsCube**(*data*, *metadata*, *processingStatus=None*, *dtype=<class 'numpy.float32'>*)
Bases: *pwspy.dataTypes._data.ICRawBase*

A class representing a single PWS acquisition. Contains methods for loading and saving to multiple formats as well as common operations used in analysis.

Parameters

- **data** – A 3-dimensional array containing the data. The dimensions should be [Y, X, Z] where X and Y are the spatial coordinates of the image and Z corresponds to the *index* dimension, e.g. wavelength, wavenumber, time, etc.
- **metadata** (*PwsMetaData*) – The metadata object associated with this data object.

- **processingStatus** (Optional[*ProcessingStatus*]) – An object that keeps track of which processing steps and corrections have been applied to this object.
- **dtype** (*type*) – the data type that the data should be stored as. The default is numpy.float32.

class ProcessingStatus(*cameraCorrected=False*, *normalizedByExposure=False*, *extraReflectionSubtracted=False*, *normalizedByReference=False*)

Bases: object

Keeps track of which processing steps have been applied to an *ICRawBase* object. By default none of these things have been done for raw data

correctCameraEffects(*correction=None*, *binning=None*)

Subtracts the darkcounts from the data. count is darkcounts per pixel. binning should be specified if it wasn't saved in the micromanager metadata. Both method arguments should be able to be loaded automatically from the metadata but for older data files they will need to be supplied manually.

Parameters

- **correction** (Optional[*CameraCorrection*]) – The cameracorrection object providing information on how to correct the data.
- **binning** (Optional[int]) – The binning that the raw data was imaged at. 2 = 2x2 binning, 3 = 3x3 binning, etc.

classmethod decodeHdf(*d*)

Load a new instance of ICRawBase from an *h5py.Dataset*

Parameters **d** (*h5py.Dataset*) – The dataset that the ICBase has been saved to

Returns data: The 3D array of *data* index: A tuple containing the *index* metadata: A dictionary containing metadata. procStatus: The processing status of the object.

Return type A tuple containing

filterDust(*kernelRadius*, *pixelSize=None*)

This method blurs the data of the PwsCube along the X and Y dimensions. This is useful if the PwsCube is being used as a reference to normalize other PwsCube. It helps blur out dust adn other unwanted small features.

Parameters

- **kernelRadius** (float) – The *sigma* of the gaussian kernel used for blurring. A greater value results in greater blurring. If *pixelSize* is provided then this is in units of *pixelSize*, otherwise it is in units of pixels.
- **pixelSize** (Optional[float]) – The size (usualy in units of microns) of each pixel in the datacube. This can generally be loaded automatically from the metadata.

Return type None

classmethod fromHdfDataset(*d*)

Load an PwsCube from an HDF5 dataset.

classmethod fromMetadata(*meta*, *lock=None*)

If provided with an PwsMetaData object this function will automatically select the correct file loading method and will return the associated PwsCube.

Parameters

- **meta** (*PwsMetaData*) – The metadata to use to load the object from.
- **lock** (Optional[Lock]) – A *Lock* object used to synchronized IO in multithreading and multiprocessing applications.

Return type *PwsCube*

Returns A new instance of *PwsCube*.

classmethod fromNano(*directory*, *metadata=None*, *lock=None*)
Loads from the file format used at NC. all data and metadata is contained in a .mat file.

Parameters

- **directory** (str) – The directory containing the data files.
- **metadata** (Optional[*PwsMetaData*]) – The metadata object associated with this acquisition
- **lock** (Optional[Lock]) – A *Lock* object used to synchronized IO in multithreading and multiprocessing applications.

Return type *PwsCube*

Returns A new instance of *PwsCube*.

classmethod fromOldPWS(*directory*, *metadata=None*, *lock=None*)
Loads from the file format that was saved by the all-matlab version of the Basis acquisition code. Data was saved in raw binary to a file called *image_cube*. Some metadata was saved to .mat files called *info2* and *info3*.

Parameters

- **directory** (str) – The directory containing the data files.
- **metadata** (Optional[*PwsMetaData*]) – The metadata object associated with this acquisition
- **lock** (Optional[Lock]) – A *Lock* object used to synchronized IO in multithreading and multiprocessing applications.

Returns A new instance of *PwsCube*.

classmethod fromTiff(*directory*, *metadata=None*, *lock=None*)
Loads from a 3D tiff file named *pws.tif*, or in some older data *MMStack.ome.tif*. Metadata can be stored in the tags of the tiff file but if there is a pwsmetadata.json file found then this is preferred. A multiprocessing.Lock object can be passed to this function so that it will acquire a lock during the hard-drive intensive parts of the function. this is useful in multi-core contexts.

Parameters

- **directory** – The directory containing the data files.
- **metadata** (Optional[*PwsMetaData*]) – The metadata object associated with this acquisition
- **lock** (Optional[Lock]) – A *Lock* object used to synchronized IO in multithreading and multiprocessing applications.

Returns A new instance of *PwsCube*.

getMeanSpectra(*mask=None*)
Calculate the average spectra within a region of the data.

Parameters **mask** (Union[*Roi*, ndarray, None]) – An optional other.Roi or boolean numpy array used to select pixels from the X and Y dimensions of the data array. If left as None then the full data array will be used as the region.

Return type Tuple[ndarray, ndarray]

Returns The average spectra within the region, the standard deviation of the spectra within the region

`static getMetadataClass()`

Return type Type[*PwsMetaData*]

Returns The metadata class associated with this subclass of ICRawBase

`classmethod loadAny(directory, metadata=None, lock=None)`
Attempt to load a *PwsCube* for any format of file in *directory*

Parameters

- **directory** (str) – The directory containing the data files.
- **metadata** (Optional[*PwsMetaData*]) – The metadata object associated with this acquisition
- **lock** (Optional[Lock]) – A *Lock* object used to synchronized IO in multithreading and multiprocessing applications.

Returns A new instance of *PwsCube*.

`normalizeByExposure()`
This is one of the first steps in most analysis pipelines. Data is divided by the camera exposure. This way two PwsCube that were acquired at different exposure times will still be on equivalent scales.

`normalizeByReference(reference)`
Normalize the raw data of this data cube by a reference cube to result in data representing arbitrarily scaled reflectance.

Parameters **reference** (PwsCube) – A reference acquisition (Usually a blank spot on a dish). The data of this acquisition will be divided by the data of the reference

`performFullPreProcessing(reference, referenceMaterial, extraReflectance, cameraCorrection=None)`
Use the *subtractExtraReflection*, *normalizeByReference*, *correctCameraEffects*, and *normalizeByExposure* methods to perform the standard pre-processing that is done before analysis.

Note: This will also end up applying corrections to the reference data. If you want to perform pre-processing on a whole batch of data then you should implement your own script based on what is done here.

Parameters

- **reference** (*self.__class__*) – A data cube to be used as a reference for normalization. Usually an image of a blank dish with cell media or air.
- **referenceMaterial** (Material) – The material that was imaged in the reference dish. The theoretically expected reflectance will be calculated assuming a "Glass/{Material}" reflective interface.
- **extraReflectance** (ExtraReflectanceCube) – The data cube containing system internal reflectance calibration information about the specific system configuration that the data was taken with.

`plotMean()`

Return type Tuple[Figure, Axes]

Returns

A figure and attached axes plotting the mean of the data along the index axis. corresponds to the mean reflectance in most cases.

selIndex(*start*, *stop*)

Parameters

- **start** (Optional[float]) – The beginning value of the index in the new object. Pass *None* to include everything.
- **stop** (Optional[float]) – The ending value of the index in the new object. Pass *None* to include everything.

Return type *PwsCube*

Returns A new instance of ICBase with only data from *start* to *stop* in the *index*.

selectLassoROI(*displayIndex=None*, *clim=None*)
Allow the user to draw a *freehand* ROI on an image of the acquisition.

Parameters **displayIndex** (Optional[int]) – Display a particular z-slice of the array for mask drawing. If *None* then the mean along Z is displayed.

Return type *Roi*

Returns An array of vertices of the polygon drawn.

selectRectangleROI(*displayIndex=None*)
Allow the user to draw a rectangular ROI on an image of the acquisition.

Parameters **displayIndex** (*int*) – is used to display a particular z-slice for mask drawing. If None then the mean along Z is displayed. Returns an array of vertices of the rectangle.

Returns An array of the 4 XY vertices of the rectangle.

Return type np.ndarray

subtractExtraReflection(*extraReflection*)
Subtract the portion of the signal that is due to internal reflections of the optical system from the data.

Parameters **extraReflection** (*ExtraReflectionCube*) – A calculated data cube indicating in units of camera counts how much of the data is from unwanted internal reflections of the system.

toHdfDataset(*g*, *name*, *fixedPointCompression=True*)
Save this object into an HDF dataset.

Parameters

- **g** (Group) – The *h5py.Group* object to create a new dataset in.
- **name** (str) – The name of the new dataset.
- **fixedPointCompression** (bool) – If True then the data will be converted from floating point to 16-bit fixed point. This results in approximately half the storage requirements at a very slight loss in precision.

Return type Group

Returns A reference to the *h5py.Group* passed in as *g*.

toOldPWS(*directory*)
Save this object to the old .mat based storage format.

Parameters **directory** – The path to the folder to save the data files to.

toTiff(*outpath*, *dtype=<class 'numpy.uint16'>*)
Save the PwsCube to the standard TIFF file format.

Parameters **outpath** (str) – The path to save the new TIFF file to.

property index: Tuple[float, ...]
Returns: The values of the datacube's index

Return type Tuple[float, ...]

property wavelengths
A tuple containing the values of the wavelengths for the data.

pwspy.dataTypes.DynCube

class pwspy.dataTypes.**DynCube**(*data*, *metadata*, *processingStatus=None*, *dtype=<class 'numpy.float32'>*)
Bases: *pwspy.dataTypes._data.ICRawBase*

A class representing a single acquisition of PWS Dynamics. In which the wavelength is held constant and the 3rd dimension of the data is time rather than wavelength. This can be analyzed to reveal information about diffusion rate. Contains methods for loading and saving to multiple formats as well as common operations used in analysis.

Parameters

- **data** – A 3-dimensional array containing the data. The dimensions should be [Y, X, Z] where X and Y are the spatial coordinates of the image and Z corresponds to the *index* dimension, e.g. wavelength, wavenumber, time, etc.
- **metadata** (*DynMetaData*) – The metadata object associated with this data object.
- **processingStatus** (Optional[*ProcessingStatus*]) – An object that keeps track of which processing steps and corrections have been applied to this object.
- **dtype** – the data type that the data should be stored as. The default is numpy.float32.

class ProcessingStatus(*cameraCorrected=False*, *normalizedByExposure=False*, *extraReflectionSubtracted=False*, *normalizedByReference=False*)
Bases: object

Keeps track of which processing steps have been applied to an *ICRawBase* object. By default none of these things have been done for raw data

correctCameraEffects(*correction=None*, *binning=None*)
Subtracts the darkcounts from the data. count is darkcounts per pixel. binning should be specified if it wasn't saved in the micromanager metadata. Both method arguments should be able to be loaded automatically from the metadata but for older data files they will need to be supplied manually.

Parameters

- **correction** (Optional[*CameraCorrection*]) – The cameracorrection object providing information on how to correct the data.
- **binning** (Optional[int]) – The binning that the raw data was imaged at. 2 = 2x2 binning, 3 = 3x3 binning, etc.

classmethod decodeHdf(*d*)
Load a new instance of ICRawBase from an *h5py.Dataset*

Parameters **d** (*h5py.Dataset*) – The dataset that the ICBase has been saved to

Returns data: The 3D array of *data* index: A tuple containing the *index* metadata: A dictionary containing metadata. procStatus: The processing status of the object.

Return type A tuple containing

filterDust(*kernelRadius*, *pixelSize=None*)

This method blurs the data of the cube along the X and Y dimensions. This is useful if the cube is being used as a reference to normalize other cube. It helps blur out dust and other unwanted small features.

Parameters

- **kernelRadius** (float) – The *sigma* of the gaussian kernel used for blurring. A greater value results in greater blurring. If *pixelSize* is provided then this is in units of *pixelSize*, otherwise it is in units of pixels.
- **pixelSize** (Optional[float]) – The size (usualy in units of microns) of each pixel in the datacube. This can generally be loaded automatically from the metadata.

classmethod fromMetadata(*meta*, *lock=None*)

Load a new instance of *DynCube* based on the information contained in a *DynMetaData* object.

Parameters

- **meta** (*DynMetaData*) – The metadata object to be used for loading.
- **lock** (Optional[Lock]) – An optional *Lock* used to synchronize IO operations in multi-threaded and multiprocessing applications.

Return type *DynCube*

Returns A new instance of *DynCube*.

classmethod fromOldPWS(*directory*, *metadata=None*, *lock=None*)

Loads from the file format that was saved by the all-matlab version of the Basis acquisition code. Data was saved in raw binary to a file called *image_cube*. Some metadata was saved to .mat files called *info2* and *info3*.

Parameters

- **directory** – The directory containing the data files.
- **metadata** (Optional[*DynMetaData*]) – The metadata object associated with this acquisition
- **lock** (Optional[Lock]) – A *Lock* object used to synchronized IO in multithreading and multiprocessing applications.

Return type *DynCube*

Returns A new instance of *DynCube*.

classmethod fromTiff(*directory*, *metadata=None*, *lock=None*)

Load a dyanmics acquisition from a tiff file. if the metadata for the acquisition has already been loaded then you can provide is as the *metadata* argument to avoid loading it again. the *lock* argument is an optional place to provide a multiprocessing.Lock which can be used when multiple files in parallel to avoid giving the hard drive too many simultaneous requests, this is probably not necessary.

Parameters

- **directory** – The directory containing the data files.
- **metadata** (Optional[*DynMetaData*]) – The metadata object associated with this acquisition
- **lock** (Optional[Lock]) – A *Lock* object used to synchronized IO in multithreading and multiprocessing applications.

Return type *DynCube*

Returns A new instance of *DynCube*.

getAutocorrelation()
Returns the autocorrelation function of dynamics data along the time axis. The ACF is calculated using fourier transforms using IFFT(FFT(data)*conj(FFT(data)))/length(data).

Return type ndarray

Returns A 3D array of the autocorrelation function of the original data.

getMeanSpectra(*mask=None*)
Calculate the average spectra within a region of the data.

Parameters **mask** (Union[*Roi*, ndarray, None]) – An optional other.Roi or boolean numpy array used to select pixels from the X and Y dimensions of the data array. If left as None then the full data array will be used as the region.

Return type Tuple[ndarray, ndarray]

Returns The average spectra within the region, the standard deviation of the spectra within the region

static getMetadataClass()

Return type Type[*DynMetaData*]

Returns The metadata class associated with this subclass of ICRawBase

classmethod loadAny(*directory*, *metadata=None*, *lock=None*)
Attempt to load a *DynCube* for any format of file in *directory*

Parameters

- **directory** (str) – The directory containing the data files.
- **metadata** (Optional[*DynMetaData*]) – The metadata object associated with this acquisition
- **lock** (Optional[Lock]) – A *Lock* object used to synchronized IO in multithreading and multiprocessing applications.

Return type *DynCube*

Returns A new instance of *DynCube*.

normalizeByExposure()
This is one of the first steps in most analysis pipelines. Data is divided by the camera exposure. This way two PwsCube that were acquired at different exposure times will still be on equivalent scales.

normalizeByReference(*reference*)
This method can accept either a DynCube (in which case it's average over time will be calculated and used for normalization) or a 2d numpy Array which should represent the average over time of a reference DynCube. The array should be 2D and its shape should match the first two dimensions of this DynCube.

Parameters **reference** (Union[*DynCube*, ndarray]) – Reference data for normalization. Usually an image of a blank piece of glass.

performFullPreProcessing(*reference*, *referenceMaterial*, *extraReflectance*, *cameraCorrection=None*)
Use the *subtractExtraReflection*, *normalizeByReference*, *correctCameraEffects*, and *normalizeByExposure* methods to perform the standard pre-processing that is done before analysis.

Note: This will also end up applying corrections to the reference data. If you want to perform pre-processing on a whole batch of data then you should implement your own script based on what is done here.

Parameters

- **reference** (*self.__class__*) – A data cube to be used as a reference for normalization. Usually an image of a blank dish with cell media or air.
- **referenceMaterial** (Material) – The material that was imaged in the reference dish. The theoretically expected reflectance will be calculated assuming a "Glass/{Material}" reflective interface.
- **extraReflectance** (ExtraReflectanceCube) – The data cube containing system internal reflectance calibration information about the specific system configuration that the data was taken with.

plotMean()

Return type Tuple[Figure, Axes]

Returns

A figure and attached axes plotting the mean of the data along the index axis. corresponds to the mean reflectance in most cases.

selIndex(*start*, *stop*)

Parameters

- **start** – The beginning value of the index in the new object. Pass *None* to include everything.
- **stop** – The ending value of the index in the new object. Pass *None* to include everything.

Return type *DynCube*

Returns A new instance of ICBase with only data from *start* to *stop* in the *index*.

selectLassoROI(*displayIndex=None*, *clim=None*)

Allow the user to draw a *freehand* ROI on an image of the acquisition.

Parameters **displayIndex** (Optional[int]) – Display a particular z-slice of the array for mask drawing. If *None* then the mean along Z is displayed.

Return type *Roi*

Returns An array of vertices of the polygon drawn.

selectRectangleROI(*displayIndex=None*)

Allow the user to draw a rectangular ROI on an image of the acquisition.

Parameters **displayIndex** (*int*) – is used to display a particular z-slice for mask drawing. If None then the mean along Z is displayed. Returns an array of vertices of the rectangle.

Returns An array of the 4 XY vertices of the rectangle.

Return type np.ndarray

subtractExtraReflection(*extraReflection*)

Subtract the portion of the signal that is due to internal reflections of the optical system from the data.

Parameters **extraReflection** (ndarray) – A calculated data cube indicating in units of camera counts how much of the data is from unwanted internal reflections of the system.

toHdfDataset(*g*, *name*, *fixedPointCompression=True*)

Save this object into an HDF dataset.

Parameters

- **g** (Group) – The *h5py.Group* object to create a new dataset in.
- **name** (str) – The name of the new dataset.
- **fixedPointCompression** (bool) – If True then the data will be converted from floating point to 16-bit fixed point. This results in approximately half the storage requirements at a very slight loss in precision.

Return type Group

Returns A reference to the *h5py.Group* passed in as *g*.

property index: Tuple[float, ...]
Returns: The values of the datacube's index

Return type Tuple[float, ...]

property times: Tuple[float, ...]
Unlike PWS where we operate along the dimension of wavelength, in dynamics we operate along the dimension of time.

Return type Tuple[float, ...]

Returns A tuple of the time values for each 2d slice along the 3rd axis of the *data* array.

pwspy.dataTypes.KCube

class pwspy.dataTypes.**KCube**(*data*, *wavenumbers*, *metadata=None*)
Bases: *pwspy.dataTypes._data.ICBase*

A class representing an PwsCube after being transformed from being described in terms of wavelength to wavenumber (k-space). Much of the analysis operated in terms of k-space.

Parameters

- **data** (ndarray) – A 3-dimensional array containing the data. The dimensions should be [Y, X, Z] where X and Y are the spatial coordinates of the image and Z corresponds to the *index* dimension, e.g. wavelength, wavenumber, time, etc.
- **wavenumbers** (Tuple[float]) – A sequence indicating the wavenumber associated with each 2D slice along the 3rd axis of the *data* array.
- **metadata** (Optional[*PwsMetaData*]) – The metadata object associated with this data object.

classmethod decodeHdf(*d*)
Load a new instance of ICBase from an *h5py.Dataset*

Parameters **d** (Dataset) – The dataset that the ICBase has been saved to

Returns (data: The 3D array of *data*, index: A tuple containing the *index*)

Return type A tuple containing

filterDust(*sigma*, *pixelSize*)
Blurs the data cube in the X and Y dimensions. Often used to remove the effects of dust on a normalization.

Parameters

- **sigma** (float) – This specifies the radius of the gaussian filter used for blurring. The units of the value are determined by *pixelSize*
- **pixelSize** (float) – The pixel size in microns. Settings this to 1 will effectively causes sigma to be in units of pixels rather than microns.

classmethod fromHdfDataset(*dataset*)
Load the KCube object from an *h5py.Dataset* in an HDF5 file

Parameters **dataset** (Dataset) – The *h5py.Dataset* that the KCube data is stored in.

Returns A new instance of this class.

Return type *KCube*

static fromOpd(*opd*, *xVals*, *useHannWindow*)
WARNING: This function is untested. it almost certainly doesn't work. Create a KCube from and opd in the form returned by KCube.getOpd. This is useful if you want to do spectral manipulation and then transform back.

classmethod fromPwsCube(*cube*)
Convert an PwsCube into a KCube. Data is converted from wavelength to wavenumber (1/lambda), interpolation is then used to linearize the data in terms of wavenumber.

Parameters **cube** (*PwsCube*) – The *PwsCube* object to generate a *KCube* object from.

Return type *KCube*

Returns A new instance of *KCube*

getAutoCorrelation(*isAutocorrMinSub*, *stopIndex*)
The autocorrelation of a signal is the covariance of a signal with a lagged version of itself, normalized so that the covariance at zero-lag is equal to 1.0 (c[0] = 1.0). The same process without normalization is the autocovariance.

A fast method for determining the autocovariance of a signal with itself is to utilize fast-fourier transforms. In this method, the signal is converted to the frequency domain using fft. The frequency-domain signal is then convolved with itself. The inverse fft is performed on this self-convolution, yielding the autocorrelation.

In this instance, the autocorrelation is determined for a series of lags, Z. Z is equal to [-P+1:P-1], where P is the quantity of measurements in each signal (the quantity of wavenumbers). Thus, the quantity of lags is equal to (2*P)-1. The fft process is fastest when performed on signals with a length equal to a power of 2. To take advantage of this property, a Z-point fft is performed on the signal, where Z is a number greater than (2*P)-1 that is also a power of 2.

Return type Tuple[ndarray, ndarray]

getMeanSpectra(*mask=None*)
Calculate the average spectra within a region of the data.

Parameters **mask** (Union[*Roi*, ndarray, None]) – An optional other.Roi or boolean numpy array used to select pixels from the X and Y dimensions of the data array. If left as None then the full data array will be used as the region.

Return type Tuple[ndarray, ndarray]

Returns The average spectra within the region, the standard deviation of the spectra within the region

getOpd(*useHannWindow*, *indexOpdStop=None*, *mask=None*)
Calculate the Fourier transform of each spectra. This can be used to get the distance (in terms of OPD) to objects that are reflecting light.

Parameters

- **useHannWindow** (bool) – If True, apply a Hann window to the data before the FFT. This reduces spectral resolution but improves dynamic range and reduces "frequency leakage".

- **indexOpdStop** (Optional[int]) – This parameter is a holdover from the original MATLAB implementation. Truncates the 3rd axis of the OPD array.
- **mask** (Optional[ndarray]) – A 2D boolean numpy array indicating which pixels should be processed.

Returns

***opd*: The 3D array of values, *opdIndex*: The sequence of OPD values associated with each** 2D slice along the 3rd axis of the *opd* data.

Return type A tuple containing

getRMSFromOPD(*lowerOPD*, *upperOPD*, *useHannWindow=False*)

Use Parseval's Theorem to calculate our signal RMS from the OPD (magnitude of fourier transform). This allows us to calculate RMS using only contributions from certain OPD ranges which ideally are correlated with a specific depth into the sample. In practice the large frequency leakage due to our limited bandwidth of measurement causes this assumption to break down, but it can still be useful if taken with a grain of salt.

Parameters

- **lowerOPD** (float) – RMS will be integrated starting at this lower limit of OPD. Note for a reflectance setup like PWS *sampleDepth = OPD / (2 * meanSampleRI)*
- **upperOPD** (float) – RMS will be integrated up to this upper OPD limit.
- **useHannWindow** (bool) – If False then use no windowing on the FFT to calculate OPD. If True then use and Hann window.

Return type ndarray

Returns A 2d numpy array of the signal RMS at each XY location in the image.

plotMean()

Return type Tuple[Figure, Axes]

Returns

A figure and attached axes plotting the mean of the data along the index axis. corresponds to the mean reflectance in most cases.

selIndex(*start*, *stop*)

Parameters

- **start** (Optional[float]) – The beginning value of the index in the new object. Pass *None* to include everything.
- **stop** (Optional[float]) – The ending value of the index in the new object. Pass *None* to include everything.

Return type Tuple[ndarray, Sequence]

Returns A new instance of ICBase with only data from *start* to *stop* in the *index*.

selectLassoROI(*displayIndex=None*, *clim=None*)

Allow the user to draw a *freehand* ROI on an image of the acquisition.

Parameters **displayIndex** (Optional[int]) – Display a particular z-slice of the array for mask drawing. If *None* then the mean along Z is displayed.

Return type *Roi*

Returns An array of vertices of the polygon drawn.

selectRectangleROI(*displayIndex=None*)
Allow the user to draw a rectangular ROI on an image of the acquisition.

Parameters **displayIndex** (*int*) – is used to display a particular z-slice for mask drawing. If None then the mean along Z is displayed. Returns an array of vertices of the rectangle.

Returns An array of the 4 XY vertices of the rectangle.

Return type np.ndarray

toHdfDataset(*g*, *name*, *fixedPointCompression=True*, *compression=None*)
Save the data of this class to a new HDF dataset.

Parameters

- **g** (*h5py.Group*) – the parent HDF Group of the new dataset.
- **name** (*str*) – the name of the new HDF dataset in group *g*.
- **fixedPointCompression** (*bool*) – if True then save the data in a special 16bit fixed-point format. Testing has shown that this has a maximum conversion error of 1.4e-3 percent. Saving is ~10% faster but requires only 50% the hard drive space.
- **compression** (Optional[str]) – The value of this argument will be passed to h5py.create_dataset for numpy arrays. See h5py documentation for available options.

Returns This is the the same h5py.Group that was passed in a *g*. It should now have a new dataset by the name of 'name'

Return type h5py.Group

property index: Tuple[float, ...]
Returns: The values of the datacube's index

Return type Tuple[float, ...]

pwspy.dataTypes.ExtraReflectanceCube

class pwspy.dataTypes.**ExtraReflectanceCube**(*data*, *wavelengths*, *metadata*)
Bases: *pwspy.dataTypes._data.ICBase*

This class represents a 3D data cube of the extra reflectance in a PWS system.

Parameters

- **data** (ndarray) – A 3D array of the extra reflectance in the system. It's values are in units of reflectance (between 0 and 1).
- **wavelengths** (Tuple[float, ...]) – The wavelengths associated with each 2D slice along the 3rd axis of the data array.
- **metadata** (*ERMetaData*) – Metadata

metadata
metadata

Type *ERMetaData*

data
data

Type ndarray

classmethod decodeHdf(*d*)
Load a new instance of ICBase from an *h5py.Dataset*

Parameters **d** (Dataset) – The dataset that the ICBase has been saved to

Returns (data: The 3D array of *data*, index: A tuple containing the *index*)

Return type A tuple containing

filterDust(*sigma*, *pixelSize*)
Blurs the data cube in the X and Y dimensions. Often used to remove the effects of dust on a normalization.

Parameters

- **sigma** (float) – This specifies the radius of the gaussian filter used for blurring. The units of the value are determined by *pixelSize*
- **pixelSize** (float) – The pixel size in microns. Settings this to 1 will effectively causes sigma to be in units of pixels rather than microns.

classmethod fromHdfDataset(*d*, *filePath=None*)
Load the ExtraReflectanceCube from *d*, an HDF5 dataset.

Parameters

- **d** (Dataset) – The *h5py.Dataset* to load the cube from.
- **filePath** (Optional[str]) – The path to the HDF file that the dataset came from.

Return type *ExtraReflectanceCube*

Returns A new instance of *ExtraReflectanceCube* loaded from HDF.

classmethod fromHdfFile(*directory*, *name*)
Load an ExtraReflectanceCube from an HDF5 file. *name* should be the file name, excluding the '_ExtraReflectance.h5' suffix.

Parameters

- **directory** (str) – The path to the folder containing the HDF file.
- **name** (str) – The *name* that the cube was saved as.

Return type *ExtraReflectanceCube*

Returns A new instance of *ExtraReflectanceCube* loaded from HDF.

classmethod fromMetadata(*md*)
Load an ExtraReflectanceCube from an ERMetaData object corresponding to an HDF5 file.

Parameters **md** (*ERMetaData*) – The metadata to be used for loading the data file.

getMeanSpectra(*mask=None*)
Calculate the average spectra within a region of the data.

Parameters **mask** (Union[*Roi*, ndarray, None]) – An optional other.Roi or boolean numpy array used to select pixels from the X and Y dimensions of the data array. If left as None then the full data array will be used as the region.

Return type Tuple[ndarray, ndarray]

Returns The average spectra within the region, the standard deviation of the spectra within the region

plotMean()

Return type Tuple[Figure, Axes]

Returns

A figure and attached axes plotting the mean of the data along the index axis. corresponds to the mean reflectance in most cases.

`selIndex`(*start*, *stop*)

Parameters

- **start** (`Optional[float]`) – The beginning value of the index in the new object. Pass *None* to include everything.
- **stop** (`Optional[float]`) – The ending value of the index in the new object. Pass *None* to include everything.

Return type `Tuple[ndarray, Sequence]`

Returns A new instance of ICBase with only data from *start* to *stop* in the *index*.

`selectLassoROI`(*displayIndex=None*, *clim=None*)

Allow the user to draw a *freehand* ROI on an image of the acquisition.

Parameters **displayIndex** (`Optional[int]`) – Display a particular z-slice of the array for mask drawing. If *None* then the mean along Z is displayed.

Return type *Roi*

Returns An array of vertices of the polygon drawn.

`selectRectangleROI`(*displayIndex=None*)

Allow the user to draw a rectangular ROI on an image of the acquisition.

Parameters **displayIndex** (*int*) – is used to display a particular z-slice for mask drawing. If None then the mean along Z is displayed. Returns an array of vertices of the rectangle.

Returns An array of the 4 XY vertices of the rectangle.

Return type np.ndarray

`toHdfDataset`(*g*)

Save the ExtraReflectanceCube to an HDF5 dataset. *g* should be an h5py Group or File.

Parameters **g** (`Group`) – The *h5py.Group* to save to.

Return type `Group`

`toHdfFile`(*directory*, *name*)

Save an ExtraReflectanceCube to an HDF5 file. The filename will be *name* with the '_ExtraReflectance.h5' suffix.

Parameters

- **directory** (`str`) – The path to the folder to save the HDF file to.
- **name** (`str`) – The *name* that the cube should be saved as.

`property index: Tuple[float, ...]`

Returns: The values of the datacube's index

Return type `Tuple[float, ...]`

`property wavelengths: Tuple[float, ...]`

Returns: The wavelengths corresponding to each element along the 3rd axis of *self.data*.

Return type `Tuple[float, ...]`

pwspy.dataTypes.ExtraReflectionCube

class pwspy.dataTypes.**ExtraReflectionCube**(*data*, *wavelengths*, *metadata*)

Bases: *pwspy.dataTypes._data.ICBase*

This class is meant to be constructed from an ExtraReflectanceCube along with additional reference measurement information. Rather than being in units of reflectance (between 0 and 1) it is in the same units as the reference measurement that is provided with, usually counts/ms or just counts.

Parameters

- **data** (ndarray) – The 3D array of the extra reflection in the system. In units of counts/ms or just counts
- **wavelengths** (Tuple[float, ...]) – The wavelengths associated with each 2D slice along the 3rd axis of the data array.
- **metadata** (*ERMetaData*) – Metadata

classmethod create(*reflectance*, *theoryR*, *reference*)

Construct and ExtraReflectionCube from an ExtraReflectanceCube and a reference measurement. The resulting ExtraReflectionCube will be in the same units as *reference*. *theoryR* should be a spectrum describing the theoretically expected reflectance of the reference data cube. Both *theoryR* and *reflectance* should be in units of reflectance (between 0 and 1).

Parameters

- **reflectance** (*ExtraReflectanceCube*) – The *ExtraReflectanceCube* to construct an *ExtraReflectionCube* from.
- **theoryR** (Series) – The theoretically predicted reflectance of material imaged in *reference*.
- **reference** (*PwsCube*) – A PWS image of a blank glass-{material} interface, usually water.

Return type *ExtraReflectionCube*

Returns A new instance of *ExtraReflectionCube*.

classmethod decodeHdf(*d*)

Load a new instance of ICBase from an *h5py.Dataset*

Parameters **d** (Dataset) – The dataset that the ICBase has been saved to

Returns (data: The 3D array of *data*, index: A tuple containing the *index*)

Return type A tuple containing

filterDust(*sigma*, *pixelSize*)

Blurs the data cube in the X and Y dimensions. Often used to remove the effects of dust on a normalization.

Parameters

- **sigma** (float) – This specifies the radius of the gaussian filter used for blurring. The units of the value are determined by *pixelSize*
- **pixelSize** (float) – The pixel size in microns. Settings this to 1 will effectively causes sigma to be in units of pixels rather than microns.

getMeanSpectra(*mask=None*)

Calculate the average spectra within a region of the data.

Parameters **mask** (Union[*Roi*, ndarray, None]) – An optional other.Roi or boolean numpy array used to select pixels from the X and Y dimensions of the data array. If left as None then the full data array will be used as the region.

Return type Tuple[ndarray, ndarray]

Returns The average spectra within the region, the standard deviation of the spectra within the region

plotMean()

Return type Tuple[Figure, Axes]

Returns

A figure and attached axes plotting the mean of the data along the index axis. corresponds to the mean reflectance in most cases.

selIndex(*start*, *stop*)

Parameters

- **start** (Optional[float]) – The beginning value of the index in the new object. Pass *None* to include everything.
- **stop** (Optional[float]) – The ending value of the index in the new object. Pass *None* to include everything.

Return type Tuple[ndarray, Sequence]

Returns A new instance of ICBase with only data from *start* to *stop* in the *index*.

selectLassoROI(*displayIndex=None*, *clim=None*)
Allow the user to draw a *freehand* ROI on an image of the acquisition.

Parameters **displayIndex** (Optional[int]) – Display a particular z-slice of the array for mask drawing. If *None* then the mean along Z is displayed.

Return type *Roi*

Returns An array of vertices of the polygon drawn.

selectRectangleROI(*displayIndex=None*)
Allow the user to draw a rectangular ROI on an image of the acquisition.

Parameters **displayIndex** (*int*) – is used to display a particular z-slice for mask drawing. If None then the mean along Z is displayed. Returns an array of vertices of the rectangle.

Returns An array of the 4 XY vertices of the rectangle.

Return type np.ndarray

toHdfDataset(*g*, *name*, *fixedPointCompression=True*, *compression=None*)
Save the data of this class to a new HDF dataset.

Parameters

- **g** (*h5py.Group*) – the parent HDF Group of the new dataset.
- **name** (*str*) – the name of the new HDF dataset in group *g*.
- **fixedPointCompression** (*bool*) – if True then save the data in a special 16bit fixed-point format. Testing has shown that this has a maximum conversion error of 1.4e-3 percent. Saving is ~10% faster but requires only 50% the hard drive space.

- **compression** (Optional[str]) – The value of this argument will be passed to h5py.create_dataset for numpy arrays. See h5py documentation for available options.

Returns This is the the same h5py.Group that was passed in a *g*. It should now have a new dataset by the name of 'name'

Return type h5py.Group

property index: Tuple[float, ...]
Returns: The values of the datacube's index

Return type Tuple[float, ...]

pwspy.dataTypes.ICBase

class pwspy.dataTypes.**ICBase**(*data*, *index*, *dtype=<class 'numpy.float32'>*)
Bases: abc.ABC

A class to handle the data operations common to PWS related *image cubes*. Does not contain any file specific functionality. uses the generic *index* attribute which can be overridden by derived classes to be wavelength, wavenumber, time, etc.

Parameters

- **data** (*np.ndarray*) – A 3-dimensional array containing the data the dimensions should be [Y, X, Z] where X and Y are the spatial coordinates of the image and Z corresponds to the *index* dimension, e.g. wavelength, wavenumber, time, etc.
- **index** (*tuple(Number)*) – A tuple containing the values of the index for the data. This could be a tuple of wavelength values, times (in the case of Dyanmics), etc.
- **dtype** (*type*) – the data type that the data should be stored as. The default is numpy.float32.

classmethod decodeHdf(*d*)
Load a new instance of ICBase from an *h5py.Dataset*

Parameters **d** (Dataset) – The dataset that the ICBase has been saved to

Returns (data: The 3D array of *data*, index: A tuple containing the *index*)

Return type A tuple containing

filterDust(*sigma*, *pixelSize*)
Blurs the data cube in the X and Y dimensions. Often used to remove the effects of dust on a normalization.

Parameters

- **sigma** (float) – This specifies the radius of the gaussian filter used for blurring. The units of the value are determined by *pixelSize*
- **pixelSize** (float) – The pixel size in microns. Settings this to 1 will effectively causes sigma to be in units of pixels rather than microns.

getMeanSpectra(*mask=None*)
Calculate the average spectra within a region of the data.

Parameters **mask** (Union[*Roi*, ndarray, None]) – An optional other.Roi or boolean numpy array used to select pixels from the X and Y dimensions of the data array. If left as None then the full data array will be used as the region.

Return type Tuple[ndarray, ndarray]

Returns The average spectra within the region, the standard deviation of the spectra within the region

plotMean()

Return type Tuple[Figure, Axes]

Returns

A figure and attached axes plotting the mean of the data along the index axis. corresponds to the mean reflectance in most cases.

selIndex(*start*, *stop*)

Parameters

- **start** (Optional[float]) – The beginning value of the index in the new object. Pass *None* to include everything.
- **stop** (Optional[float]) – The ending value of the index in the new object. Pass *None* to include everything.

Return type Tuple[ndarray, Sequence]

Returns A new instance of ICBase with only data from *start* to *stop* in the *index*.

selectLassoROI(*displayIndex=None*, *clim=None*)

Allow the user to draw a *freehand* ROI on an image of the acquisition.

Parameters **displayIndex** (Optional[int]) – Display a particular z-slice of the array for mask drawing. If *None* then the mean along Z is displayed.

Return type *Roi*

Returns An array of vertices of the polygon drawn.

selectRectangleROI(*displayIndex=None*)

Allow the user to draw a rectangular ROI on an image of the acquisition.

Parameters **displayIndex** (*int*) – is used to display a particular z-slice for mask drawing. If None then the mean along Z is displayed. Returns an array of vertices of the rectangle.

Returns An array of the 4 XY vertices of the rectangle.

Return type np.ndarray

toHdfDataset(*g*, *name*, *fixedPointCompression=True*, *compression=None*)

Save the data of this class to a new HDF dataset.

Parameters

- **g** (*h5py.Group*) – the parent HDF Group of the new dataset.
- **name** (*str*) – the name of the new HDF dataset in group *g*.
- **fixedPointCompression** (*bool*) – if True then save the data in a special 16bit fixed-point format. Testing has shown that this has a maximum conversion error of 1.4e-3 percent. Saving is ~10% faster but requires only 50% the hard drive space.
- **compression** (Optional[str]) – The value of this argument will be passed to h5py.create_dataset for numpy arrays. See h5py documentation for available options.

Returns This is the the same h5py.Group that was passed in a *g*. It should now have a new dataset by the name of 'name'

Return type h5py.Group

property index: Tuple[float, ...]
Returns: The values of the datacube's index

Return type Tuple[float, ...]

pwspy.dataTypes.ICRawBase

class pwspy.dataTypes.**ICRawBase**(*data*, *index*, *metadata*, *processingStatus=None*, *dtype=<class 'numpy.float32'>*)
Bases: *pwspy.dataTypes._data.ICBase*, abc.ABC

This class represents data cubes which are not derived from other data cubes. They represent raw acquired data that exists as data files on the computer. For this reason they may need to have hardware specific corrections applied to them such as normalizing out exposure time, linearizing camera counts, subtracting dark counts, etc. The most important change is the addition of *metadata*.

Parameters

- **data** (*np.ndarray*) – A 3-dimensional array containing the data. The dimensions should be [Y, X, Z] where X and Y are the spatial coordinates of the image and Z corresponds to the *index* dimension, e.g. wavelength, wavenumber, time, etc.
- **index** (*tuple*) – A tuple containing the values of the index for the data. This could be a tuple of wavelength values, times (in the case of Dynamics), etc.
- **metadata** (*pwsdtmd.MetaDataBase*) – The metadata object associated with this data object.
- **processingStatus** (ProcessingStatus) – An object that keeps track of which processing steps and corrections have been applied to this object.
- **dtype** (*type*) – the data type that the data should be stored as. The default is numpy.float32.

class ProcessingStatus(*cameraCorrected=False*, *normalizedByExposure=False*, *extraReflectionSubtracted=False*, *normalizedByReference=False*)
Bases: object

Keeps track of which processing steps have been applied to an *ICRawBase* object. By default none of these things have been done for raw data

correctCameraEffects(*correction=None*, *binning=None*)
Subtracts the darkcounts from the data. count is darkcounts per pixel. binning should be specified if it wasn't saved in the micromanager metadata. Both method arguments should be able to be loaded automatically from the metadata but for older data files they will need to be supplied manually.

Parameters

- **correction** (Optional[*CameraCorrection*]) – The cameracorrection object providing information on how to correct the data.
- **binning** (Optional[int]) – The binning that the raw data was imaged at. 2 = 2x2 binning, 3 = 3x3 binning, etc.

classmethod decodeHdf(*d*)
Load a new instance of ICRawBase from an *h5py.Dataset*

Parameters **d** (*h5py.Dataset*) – The dataset that the ICBase has been saved to

Returns data: The 3D array of *data* index: A tuple containing the *index* metadata: A dictionary containing metadata. procStatus: The processing status of the object.

Return type A tuple containing

filterDust(*sigma*, *pixelSize*)
Blurs the data cube in the X and Y dimensions. Often used to remove the effects of dust on a normalization.

Parameters

- **sigma** (float) – This specifies the radius of the gaussian filter used for blurring. The units of the value are determined by *pixelSize*
- **pixelSize** (float) – The pixel size in microns. Settings this to 1 will effectively causes sigma to be in units of pixels rather than microns.

getMeanSpectra(*mask=None*)
Calculate the average spectra within a region of the data.

Parameters **mask** (Union[*Roi*, ndarray, None]) – An optional other.Roi or boolean numpy array used to select pixels from the X and Y dimensions of the data array. If left as None then the full data array will be used as the region.

Return type Tuple[ndarray, ndarray]

Returns The average spectra within the region, the standard deviation of the spectra within the region

abstract static getMetadataClass()

Return type Type[MetaDataBase]

Returns The metadata class associated with this subclass of ICRawBase

normalizeByExposure()
This is one of the first steps in most analysis pipelines. Data is divided by the camera exposure. This way two PwsCube that were acquired at different exposure times will still be on equivalent scales.

abstract normalizeByReference(*reference*)
Normalize the raw data of this data cube by a reference cube to result in data representing arbitrarily scaled reflectance.

Parameters **reference** (*self.__class__*) – A reference acquisition. Usually an image taken from a blank piece of glass.

performFullPreProcessing(*reference*, *referenceMaterial*, *extraReflectance*, *cameraCorrection=None*)
Use the *subtractExtraReflection*, *normalizeByReference*, *correctCameraEffects*, and *normalizeByExposure* methods to perform the standard pre-processing that is done before analysis.

Note: This will also end up applying corrections to the reference data. If you want to perform pre-processing on a whole batch of data then you should implement your own script based on what is done here.

Parameters

- **reference** (*self.__class__*) – A data cube to be used as a reference for normalization. Usually an image of a blank dish with cell media or air.
- **referenceMaterial** (Material) – The material that was imaged in the reference dish. The theoretically expected reflectance will be calculated assuming a "Glass/{Material}" reflective interface.
- **extraReflectance** (ExtraReflectanceCube) – The data cube containing system internal reflectance calibration information about the specific system configuration that the data was taken with.

plotMean()

Return type Tuple[Figure, Axes]

Returns

A figure and attached axes plotting the mean of the data along the index axis. corresponds to the mean reflectance in most cases.

`selIndex`(*start*, *stop*)

Parameters

- **start** (`Optional[float]`) – The beginning value of the index in the new object. Pass *None* to include everything.
- **stop** (`Optional[float]`) – The ending value of the index in the new object. Pass *None* to include everything.

Return type `Tuple[ndarray, Sequence]`

Returns A new instance of ICBase with only data from *start* to *stop* in the *index*.

`selectLassoROI`(*displayIndex=None*, *clim=None*)

Allow the user to draw a *freehand* ROI on an image of the acquisition.

Parameters **displayIndex** (`Optional[int]`) – Display a particular z-slice of the array for mask drawing. If *None* then the mean along Z is displayed.

Return type *Roi*

Returns An array of vertices of the polygon drawn.

`selectRectangleROI`(*displayIndex=None*)

Allow the user to draw a rectangular ROI on an image of the acquisition.

Parameters **displayIndex** (*int*) – is used to display a particular z-slice for mask drawing. If None then the mean along Z is displayed. Returns an array of vertices of the rectangle.

Returns An array of the 4 XY vertices of the rectangle.

Return type np.ndarray

`abstract subtractExtraReflection`(*extraReflection*)

Subtract the portion of the signal that is due to internal reflections of the optical system from the data.

Parameters **extraReflection** (*ExtraReflectionCube*) – A calculated data cube indicating in units of camera counts how much of the data is from unwanted internal reflections of the system.

`toHdfDataset`(*g*, *name*, *fixedPointCompression=True*)

Save this object into an HDF dataset.

Parameters

- **g** (`Group`) – The *h5py.Group* object to create a new dataset in.
- **name** (`str`) – The name of the new dataset.
- **fixedPointCompression** (`bool`) – If True then the data will be converted from floating point to 16-bit fixed point. This results in approximately half the storage requirements at a very slight loss in precision.

Return type `Group`

Returns A reference to the *h5py.Group* passed in as *g*.

`property index: Tuple[float, ...]`

Returns: The values of the datacube's index

Return type Tuple[float, ...]

2.2.3 Other Classes

Roi	This class represents a single Roi used to select a specific region of an image.
RoiFile	This class represents a single Roi File used to save and load an ROI.
CameraCorrection	This class represents all the information needed to correct camera related hardware defects in our data.
Acquisition	This class handles the file structure of a single acquisition.
FluorescenceImage	Represents a fluorescence image taken by the PWS acquisition software.

pwspy.dataTypes.Roi

class pwspy.dataTypes.**Roi**(*mask*, *verts*)

Bases: object

This class represents a single Roi used to select a specific region of an image. The Roi consists of a *mask* (a boolean array specifying which pixels are included in the Roi), a set of of *vertices* (a 2 x N array specifying the vertices of the polygon enclosing the mask, this is useful if you want to adjust the Roi later. Rather than calling the constructor directly you will generally create one of these objects through one of the *class methods* that construct one for you.

Parameters

- **mask** (ndarray) – A 2D boolean array where the True values indicate pixels that are within the ROI.
- **verts** (Union[ndarray, Polygon]) – Can be a sequence of 2D (x, y) coordinates indicating the border of the ROI or a shapely *Polygon*. If an array of coordinates is used then it will be converted to the shell of a shapely polygon internally. While this information is partially redundant with the mask it is useful for many applications and can be complicated to calculate from *mask*.

classmethod fromMask(*mask*)

Use rasterio to create find the vertices of a mask. :type mask: ndarray :param mask: A boolean array. The mask have only one contiguous *True* region

Return type *Roi*

Returns A new instance of *Roi*

classmethod fromVerts(*verts*, *dataShape*)

Automatically generate the mask for an Roi using just the vertices of an enclosing polygon.

Parameters

- **verts** (ndarray) – A sequence of 2D (x, y) coordinates indicating the border of the ROI.
- **dataShape** (Tuple[float, float]) – A tuple giving the shape of the array that this Roi is associated with.

Return type *Roi*

Returns A new instance of *Roi*

Examples

myRoi = Roi.fromVerts('nucleus', 1, polyVerts, (1024, 1024))

transform(*matrix*)
Return a copy of this Roi that has been transformed by an affine transform matrix like the one returned by opencv.estimateRigidTransform. This can be obtained using the functions in the utility.machineVision module.

Parameters **matrix** (ndarray) – A 2x3 numpy array representing an affine transformation.

Return type *Roi*

Returns A new instance of Roi representing this Roi after transformation.

property verts: numpy.ndarray
An array of vertices for the outer ring of the polygon. For most ROIs they only have an outer ring anyway.

Return type ndarray

pwspy.dataTypes.RoiFile

class pwspy.dataTypes.**RoiFile**(*name*, *number*, *roi*, *filePath*, *fileFormat*, *acquisition*)
Bases: object

This class represents a single Roi File used to save and load an ROI. Each Roi File is identified by a *name* and a *number*. The recommended file format is HDF2, in this format multiple rois of the same name but differing numbers can be saved in a single HDF file.

Parameters

- **name** (str) – The name used to identify this ROI. Multiple ROIs can share the same name but must have unique numbers.
- **number** (int) – The number used to identify this ROI. Each ROI with the same name must have a unique number.
- **roi** (*Roi*) – The ROI object associated with this file.
- **filePath** (str) – The path to the file that this object was loaded from.
- **fileFormat** (*FileFormats*) – The format of the file that this object was loaded from.
- **acquisition** (*Acquisition*) – The acquisition object that this ROI belongs to.

class FileFormats(*value*)
Bases: enum.Enum

An enumerator of the different file formats that an ROI can be saved to.

delete()
Delete the dataset associated with the Roi object.

static deleteRoi(*directory*, *name*, *number*, *fformat=None*)
Delete the dataset associated with the Roi object specified by *name* and *num*.

Parameters

- **directory** (str) – The path to the folder containing the Roi file.
- **name** (str) – The name used to identify this ROI.
- **number** (int) – The number used to identify this ROI.
- **fformat** (Optional[*FileFormats*]) – The format of the file.

Raises **FileNotFoundError** – If the file isn't found.

classmethod fromHDF(*directory*, *name*, *number*, *acquisition=None*)
Load an Roi from the newest ROI format of HDF file.

Parameters

- **directory** (str) – The path to the directory containing the HDF file.
- **name** (str) – The name used to identify this ROI.
- **number** (int) – The number used to identify this ROI.

Raises **OSError** – If the file was not found

Return type *RoiFile*

Returns A new instance of Roi loaded from file

Examples

myRoi = Roi.fromHDF('~/Desktop', 'nucleus', 1)

classmethod fromHDF_legacy(*directory*, *name*, *number*, *acquisition=None*)
Load an Roi from an HDF file. Uses the old HDF2 format.

Parameters

- **directory** (str) – The path to the directory containing the HDF file.
- **name** (str) – The name used to identify this ROI.
- **number** (int) – The number used to identify this ROI.

Raises **OSError** – If the file was not found

Return type *RoiFile*

Returns A new instance of Roi loaded from file

Examples

myRoi = Roi.fromHDF('~/Desktop', 'nucleus', 1)

classmethod fromHDF_legacy_legacy(*directory*, *name*, *number*, *acquisition=None*)
Load an Roi from an older version of the HDF file format which did not include the vertices parameter.

Parameters

- **directory** (str) – The path to the directory containing the HDF file.
- **name** (str) – The name used to identify this ROI.
- **number** (int) – The number used to identify this ROI.

Raises **OSError** – If the file was not found

Return type *RoiFile*

Returns A new instance of Roi loaded from file

classmethod fromMat(*directory*, *name*, *number*, *acquisition=None*)
Load an Roi from a .mat file saved in matlab. This file format is not recommended as it does not include the *vertices* parameter which is useful for visually rendering and readjusting the Roi.

Parameters

- **directory** (str) – The path to the directory containing the HDF file.
- **name** (str) – The name used to identify this ROI.
- **number** (int) – The number used to identify this ROI.

Return type *RoiFile*

Returns A new instance of Roi loaded from file

getRoi()
Return the ROI object associated with this file.

Return type *Roi*

Returns The *Roi* object containing geometry information.

static getValidRoisInPath(*path*)
Search the *path* for valid roiFile files and return the detected rois as a list of tuple where each tuple contains the *name*, *number*, and file format for the Roi.

Parameters **path** (str) – The path to the folder containing the Roi files.

Returns name: The detected Roi name number: The detected Roi number fformat: The file format of the file that the Roi is stored in

Return type A list of tuples containing

classmethod loadAny(*directory*, *name*, *number*, *acquisition=None*)
Attempt loading any of the known file formats.

Parameters

- **directory** (str) – The path to the directory containing the HDF file.
- **name** (str) – The name used to identify this ROI.
- **number** (int) – The number used to identify this ROI.

Return type *RoiFile*

Returns A new instance of Roi loaded from file

classmethod toHDF(*roi*, *name*, *number*, *directory*, *overwrite=False*, *acquisition=None*)
Save the Roi to an HDF file in the specified directory. The filename is automatically chosen based on the *name* parameter of the Roi. Multiple Roi's with the same *name* will be saved into the same file if they have differing *number* parameters. If *overwrite* is true then any existing dataset will be replaced, otherwise an error will be raised.

Parameters

- **roi** (*Roi*) – The ROI to save.
- **name** (str) – The name name to save as. This will be part of the file name
- **number** (int) – The ROI number to save as. Multiple ROIS of the same name can be saved to the same file but the numbers must be unique
- **directory** (str) – The path of the folder to save the new HDF file to. The file will be named automatically based on the *name* attribute of the Roi
- **overwrite** (Optional[bool]) – If True then if an Roi with the same *number* as this Roi is found it will be overwritten.

Return type *RoiFile*

update(*roi*)
Save a new roiFile to the existing file. :type roi: *Roi* :param roi: The updated ROI to save

pwspy.dataTypes.CameraCorrection

class pwspy.dataTypes.**CameraCorrection**(*darkCounts*, *linearityPolynomial=None*)
Bases: object

This class represents all the information needed to correct camera related hardware defects in our data. This includes a dark count value (The counts registered when no light is incident on the camera. It also includes a polynomial that is used to linearize the counts. E.G. if you image something over a range of exposure times you would expect the measured counts to be proportional to the exposure time. However on some cameras this is not the case.

darkCounts
Dark count for a single pixel of the camera. This will be subtracted from the data in pre-processing. When binning is used the dark counts are summed together, so if you measure a dark count of 400 with 2x2 binning then the single pixel dark count is 100.

Type float

linearityPolynomial
Sequence of polynomial coefficients [a,b,c,etc...] in the order a*x + b*x^2 + c*x^3 + etc... Used to linearize the counts from the camera so that they are linearly proportional to the image brightness. This can generally be left as *None* for sCMOS cameras but it is often required for CCD type cameras.

Type t_.Sequence[float, ...]

classmethod fromJsonFile(*filePath*)
Load the camera correction from a json text file.

Parameters **filePath** (str) – The file path of the JSON file to load from.

Return type *CameraCorrection*

Returns A new instance of *CameraCorrection*.

Examples

corr = CameraCorrection.fromJsonFile('~/Desktop/camera.json')

toJsonFile(*filePath*)
Save the camera correction to a JSON formatted text file.

Parameters **filePath** (str) – The file path for the new JSON file.

pwspy.dataTypes.Acquisition

class pwspy.dataTypes.**Acquisition**(*directory*)
Bases: object

This class handles the file structure of a single acquisition. this can include a PWS acquisition as well as colocalized Dynamics and fluorescence.

Parameters **directory** (Union[str, PathLike]) – the file path the root directory of the acquisition

editNotes()
Create a *notes.txt* file if it doesn't already exists and open it in a text editor.

getNotes()
Return the contents of *notes.txt* as a string.

Return type str

getRois()
Return information about the Rois found in the acquisition's file path. See documentation for Roi.getValidRoisInPath()

Return type List[Tuple[str, int, *FileFormats*]]

getThumbnail()
Return a thumbnail from any of the available acquisitions. Should be an 8bit normalized image.

Return type ndarray

hasNotes()
Indicates whether or not a *notes.txt* file was found.

Return type bool

loadRoi(*name*, *num*, *fformat=None*)
Load a Roi that has been saved to file in the acquisition's file path.

Return type *RoiFile*

saveRoi(*roiName*, *roiNumber*, *roi*, *overwrite=False*)
Save a Roi to file in the acquisition's file path.

Parameters

- **roiName** (str) – The name to identify this ROI
- **roiNumber** (int) – The number to identify this ROI
- **roi** (*Roi*) – The ROI object defining the ROI geometry
- **overwrite** (bool) – If True then any existing ROIFile matching this name and number will be overwritten. Otherwise an OSError is raised.

Return type *RoiFile*

Returns A reference to the new ROIFile

Raises **OSError** – If *overwrite* is False and an ROI of the same name and number

already exists then an OSError will be raised.

dynamics
Returns None if no dynamics acquisition was found.

Type *DynMetaData*

fluorescence
Newer acquisitions allow for multiple fluorescence images saved to numbered subfolders

Type List[*FluorMetaData*]

pws
Returns None if no PWS acquisition was found.

Type *PwsMetaData*

pwspy.dataTypes.FluorescenceImage

class pwspy.dataTypes.**FluorescenceImage**(*data*, *md*)
Bases: object

Represents a fluorescence image taken by the PWS acquisition software.

Parameters

- **data** (ndarray) – A 2D array of image data.
- **md** (*FluorMetaData*) – The metadata object associated with this image.

classmethod fromMetadata(*md*, *lock=None*)
Load an image from the metadata object.

Parameters

- **md** (*FluorMetaData*) – The metadata object to load the image from.
- **lock** (Optional[Lock]) – An optional multiprocessing *Lock* object to synchronize file access in multiprocessing contexts

Return type *FluorescenceImage*

Returns A new instance of *FluorescenceImage*.

classmethod fromTiff(*directory*, *acquisitionDirectory=None*)
Load an image from a TIFF file.

Parameters

- **directory** (str) – The path to the folder containing the TIFF file.
- **acquisitionDirectory** (Optional[*Acquisition*]) – The *Acquisition* object associated with this acquisition.

Return type *FluorescenceImage*

Returns A new instanse of *FluorescenceImage*.

toTiff(*directory*)
Save this object to a TIFF file.

Parameters **directory** (str) – The path to the folder to save the new file to.

2.2.4 Inheritance

2.3 pwspy.utility

Useful subpackages ranging many different topics

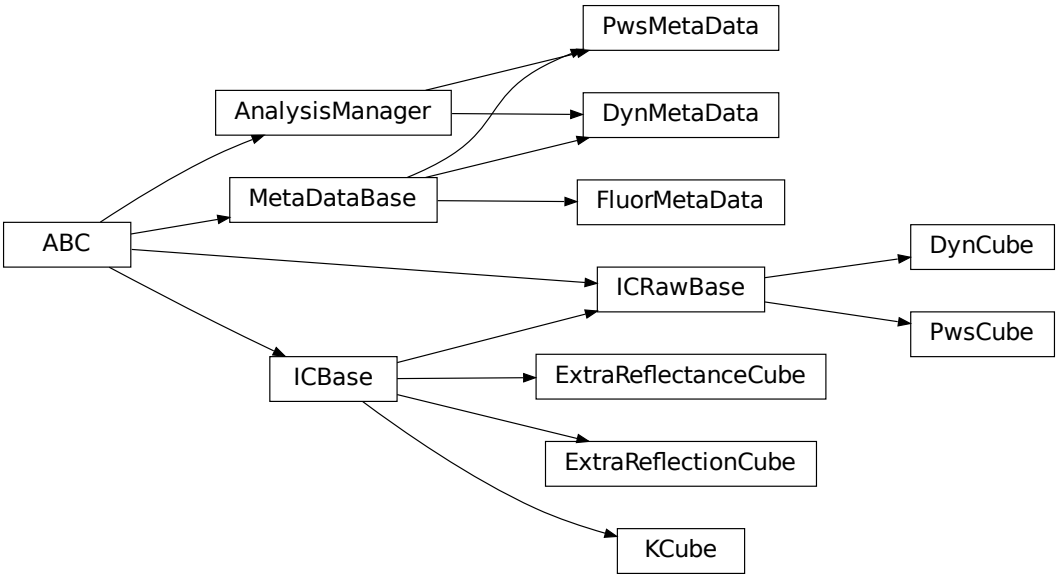


Fig. 1: Abstract base classes define common behavior between the implementations of the various data types making it easy to write software using PWSpy that will work for all available data type implementations.

2.3.1 Modules

acquisition	This package includes functionality for loading experiment information saved by the PWS "Acquisition Sequencer" which is part of the PWS plugin for Micro-Manager.
DConversion	Classes This class is used to connect to the MATLAB Sigma to D conversion library.
fileIO	Functions for quickly loading files using parallel processing.
fluorescence	Functions related to fluorescent images.
machineVision	Useful functions for processing images.
micromanager	Objects useful for dealing with files saved by Micro-Manager.
misc	Objects that are generally useful in python programming.
plotting	Image Plotting
reflection	A package containing functionality useful for calculation reflections.

pwspy.utility.acquisition

This package includes functionality for loading experiment information saved by the PWS "Acquisition Sequencer" which is part of the PWS plugin for Micro-Manager.

Functions

Classes

RuntimeSequenceSettings(uuid, dateString, ...)	This represents the object saved when a new acquisition is run.
SequenceAcquisition(acquisition)	An object linking and acquisition with a sequencerCoordinate file.
SequencerCoordinate(coordSteps, uuid)	A coordinate that fully defines a position within a *tree* of steps.
SequencerCoordinateRange(coordSteps)	A coordinate that can have multiple iterations selected at once.
SequencerStep(id, settings, stepType[, children])	Implementation of a TreeItem for representing a sequencer step.
IterableSequencerStep(*args, **kwargs)	A base-class for steps which are iterable.
ZStackStep(*args, **kwargs)	
TimeStep(*args, **kwargs)	
PositionsStep(*args, **kwargs)	
ContainerStep(id, settings, stepType[, children])	A class for steps which can contain other steps within it.

pwspy.utility.acquisition.RuntimeSequenceSettings

class pwspy.utility.acquisition.**RuntimeSequenceSettings**(*uuid*, *dateString*, *rootStep*)

Bases: object

This represents the object saved when a new acquisition is run.

pwspy.utility.acquisition.SequenceAcquisition

class pwspy.utility.acquisition.**SequenceAcquisition**(*acquisition*)

Bases: object

An object linking and acquisition with a sequencerCoordinate file.

acquisition
The pwspy.dataTypes.Acquisition object linking to the raw data

sequencerCoordinate
The coordinate object locating the acquisition within a sequence.

pwspy.utility.acquisition.SequencerCoordinate

class pwspy.utility.acquisition.**SequencerCoordinate**(*coordSteps*, *uuid*)

Bases: object

A coordinate that fully defines a position within a *tree* of steps.

Parameters

- **coordSteps** (Sequence[Tuple[int, int]]) – A sequence of tuples of the form (stepId, stepIteration) where *stepId* is the id number of the step being referred to. If the step is an

iterable step (multiple position, timeseries, etc.) then *stepIteration* should indicate the iteration number, otherwise it should be *None*.

- **uuid** (str) – A universally unique ID string associated with the run of the sequencer that this coordinate is associated with.

getStepIteration(*step*)

Parameters **step** (Union[int, *SequencerStep*]) – May be the ID number of the step or a reference to the actual step.

Return type Optional[int]

Returns The iteration of *Step* that this coordinate corresponds to. If the step is not an iterable step then *None* will be returned.

isSubPathOf(*other*)

Check if *self* is a parent path of the *item* coordinate

pwspy.utility.acquisition.SequencerCoordinateRange

class pwspy.utility.acquisition.**SequencerCoordinateRange**(*coordSteps*)

Bases: object

A coordinate that can have multiple iterations selected at once.

Parameters **coordSteps** (Sequence[Tuple[int, Optional[Sequence[int]]]]) – A sequence of tuples where each tuple represents an acceptable coordinate range for each step in the tree path. Tuple is of the form (ID, iterations) where ID is an integer referring to the id number of the step and iterations indicates the iterations of that step that are considered in range. *iterations* can be *None* or a sequence of *ints* that are considered in range. If *iterations is `None* then all iterations are considered in range. *None* is also the only acceptable value for steps which do not iterate.

setAcceptedIterations(*stepId*, *iterations*)

Sets the acceptable iterations for the step associated with *stepId*

Parameters

- **stepId** (int) – The id of the step you want to adjust the iteration range for
- **iterations** (Optional[Sequence[int]]) – A sequence if iteration numbers to include. If *None* then all iterations are accepted.

pwspy.utility.acquisition.SequencerStep

class pwspy.utility.acquisition.**SequencerStep**(*id*, *settings*, *stepType*, *children=None*)

Bases: pwspy.utility.acquisition._treeItem.TreeItem

Implementation of a TreeItem for representing a sequencer step.

Parameters

- **id** (int) – The unique integer assigned to this step by the acquisition software
- **settings** (dict) – The settings for this step. Saved as JSON in the sequence file.
- **stepType** (str) – Indicates what type of step this is. E.g. acquisition, time-series, focus lock, etc.

- **children** (Optional[List[*SequencerStep*]]) – A list of steps which are direct children of this step.

getCoordinate()
Returns a sequencer coordinate range that points to this steps location in the tree of steps.

Return type *SequencerCoordinateRange*

getTreePath()
Return a list of steps starting with the root step and ending with this step.

Return type Tuple[TreeItem]

static hook(*dct*)
This method defines how the JSON library should translate from JSON to one of these objects. :type dct: dict :param dct: The *dict* representing the raw representation of the JSON

iterateChildren()
Recursively iterate through all children of this step

printSubTree(*_indent=0*)
Print out this step and all sub-steps in a human-readable format.

Return type None

row()
Return which row we are with respect to the parent.

Return type int

pwspy.utility.acquisition.IterableSequencerStep

class pwspy.utility.acquisition.**IterableSequencerStep**(**args*, ***kwargs*)
Bases: *pwspy.utility.acquisition.steps.ContainerStep*

A base-class for steps which are iterable. Despite only being a single step they run multiple times in an acquisition. This add some complications as we want to keep track of which iteration the sub-steps of this belong to.

getCoordinate()
Returns a sequencer coordinate range that points to this steps location in the tree of steps.

Return type *SequencerCoordinateRange*

abstract getIterationName(*iteration*)
Return the name associated with *iteration* E.G. for a multiple-positions step this will be the name assigned to the position in the position list. :type iteration: int :param iteration: The iteeration number we are interested in.

Returns: A name for the requested iteration.

Return type str

getTreePath()
Return a list of steps starting with the root step and ending with this step.

Return type Tuple[TreeItem]

static hook(*dct*)
This method defines how the JSON library should translate from JSON to one of these objects. :type dct: dict :param dct: The *dict* representing the raw representation of the JSON

iterateChildren()
Recursively iterate through all children of this step

printSubTree(*_indent=0*)
Print out this step and all sub-steps in a human-readable format.

Return type None

row()
Return which row we are with respect to the parent.

Return type int

abstract stepIterations()
Return the total number of iterations of this step.

pwspy.utility.acquisition.ZStackStep

class pwspy.utility.acquisition.**ZStackStep**(**args*, ***kwargs*)
Bases: *pwspy.utility.acquisition.steps.IterableSequencerStep*

getCoordinate()
Returns a sequencer coordinate range that points to this steps location in the tree of steps.

Return type *SequencerCoordinateRange*

getIterationName(*iteration*)
Return the name associated with *iteration* E.G. for a multiple-positions step this will be the name assigned to the position in the position list. :type iteration: int :param iteration: The iteeration number we are interested in.

Returns: A name for the requested iteration.

Return type str

getTreePath()
Return a list of steps starting with the root step and ending with this step.

Return type Tuple[TreeItem]

static hook(*dct*)
This method defines how the JSON library should translate from JSON to one of these objects. :type dct: dict :param dct: The *dict* representing the raw representation of the JSON

iterateChildren()
Recursively iterate through all children of this step

printSubTree(*_indent=0*)
Print out this step and all sub-steps in a human-readable format.

Return type None

row()
Return which row we are with respect to the parent.

Return type int

stepIterations()
Return the total number of iterations of this step.

pwspy.utility.acquisition.TimeStep

class pwspy.utility.acquisition.**TimeStep**(**args*, ***kwargs*)

Bases: *pwspy.utility.acquisition.steps.IterableSequencerStep*

getCoordinate()

Returns a sequencer coordinate range that points to this steps location in the tree of steps.

Return type *SequencerCoordinateRange*

getIterationName(*iteration*)

Return the name associated with *iteration* E.G. for a multiple-positions step this will be the name assigned to the position in the position list. :type iteration: `int` :param iteration: The iteeration number we are interested in.

Returns: A name for the requested iteration.

Return type `str`

getTreePath()

Return a list of steps starting with the root step and ending with this step.

Return type `Tuple[TreeItem]`

static hook(*dct*)

This method defines how the JSON library should translate from JSON to one of these objects. :type dct: `dict` :param dct: The *dict* representing the raw representation of the JSON

iterateChildren()

Recursively iterate through all children of this step

printSubTree(*_indent=0*)

Print out this step and all sub-steps in a human-readable format.

Return type `None`

row()

Return which row we are with respect to the parent.

Return type `int`

stepIterations()

Return the total number of iterations of this step.

pwspy.utility.acquisition.PositionsStep

class pwspy.utility.acquisition.**PositionsStep**(**args*, ***kwargs*)

Bases: *pwspy.utility.acquisition.steps.IterableSequencerStep*

getCoordinate()

Returns a sequencer coordinate range that points to this steps location in the tree of steps.

Return type *SequencerCoordinateRange*

getIterationName(*iteration*)

Return the name associated with *iteration* E.G. for a multiple-positions step this will be the name assigned to the position in the position list. :type iteration: `int` :param iteration: The iteeration number we are interested in.

Returns: A name for the requested iteration.

Return type `str`

getTreePath()
Return a list of steps starting with the root step and ending with this step.

Return type Tuple[TreeItem]

static hook(*dct*)
This method defines how the JSON library should translate from JSON to one of these objects. :type dct: dict :param dct: The *dict* representing the raw representation of the JSON

iterateChildren()
Recursively iterate through all children of this step

printSubTree(*_indent=0*)
Print out this step and all sub-steps in a human-readable format.

Return type None

row()
Return which row we are with respect to the parent.

Return type int

stepIterations()
Return the total number of iterations of this step.

pwspy.utility.acquisition.ContainerStep

class pwspy.utility.acquisition.**ContainerStep**(*id*, *settings*, *stepType*, *children=None*)
Bases: *pwspy.utility.acquisition.steps.SequencerStep*

A class for steps which can contain other steps within it.

getCoordinate()
Returns a sequencer coordinate range that points to this steps location in the tree of steps.

Return type *SequencerCoordinateRange*

getTreePath()
Return a list of steps starting with the root step and ending with this step.

Return type Tuple[TreeItem]

static hook(*dct*)
This method defines how the JSON library should translate from JSON to one of these objects. :type dct: dict :param dct: The *dict* representing the raw representation of the JSON

iterateChildren()
Recursively iterate through all children of this step

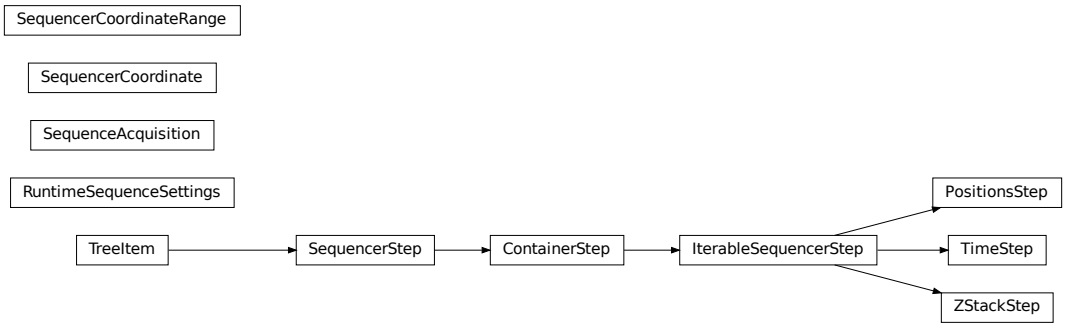
printSubTree(*_indent=0*)
Print out this step and all sub-steps in a human-readable format.

Return type None

row()
Return which row we are with respect to the parent.

Return type int

Inheritance



pwspy.utility.DConversion

Classes

This class is used to connect to the MATLAB Sigma to D conversion library.

S2DMatlabBridge(s2dPath)	Opens a MATLAB process to run the Sigma2D conversion code.

pwspy.utility.DConversion.S2DMatlabBridge

class pwspy.utility.DConversion.**S2DMatlabBridge**(*s2dPath*)

Bases: object

Opens a MATLAB process to run the Sigma2D conversion code. You must have the MATLAB engine for Python installed in your Python environment.

Parameters **s2dPath** (str) – The file path to the SigmaConversion MATLAB package.

SigmaToD_AllInputs(*sigmaIn*, *system_config*, *Nf*, *thickIn*)

Run the complete sigma to D conversion function.

Parameters

- **sigmaIn** (Union[Sequence[float], float]) – The sigma values you want to convert.
- **system_config** (*S2D.SystemConfiguration*) – The SystemConfiguration object to use.
- **Nf** (float) – The genomic length of a packing domain.
- **thickIn** (float) – The expected thickness of the sample.

Returns This is analogous to *D_b* in Aya's paper. The *model parameter*. dCorrected: This is the true *D*. This is usually what we care about. Nf_expected: The genomic length we expect

based on D and the calculated lMax. lmax_corrected: Calculation of LMax from Nf and Db based on eqn. 2.

Return type dOut

`close()`
Close the MATLAB engine. This may not be necessary as the engine will be automatically closed when Python shuts down.

`createRIDefinitionFromGladstoneDale`(*mediaRI*, *CVC*)
Create an S2D.RIDefinition object from the GladstoneDale equation.

Parameters

- **mediaRI** (`float`) – The refractive index of the media that the chromatin is immersed in.
- **CVC** (`float`) – The CVC, also referred to as Phi. The ratio of Chromatin Volume : Total Volume, kind of like density.

Return type matlab.object

Returns A S2D.RIDefinition object.

`createSystemConfiguration`(*ri_def*, *na_i*, *na_c*, *center_lambda*, *oil_immersion*, *cell_glass_interface*)
Create a system configuration object. This object contains all information about the microscope and sample refractive indices.

Parameters

- **ri_def** (*S2D.RIDefinition*) – An RI definition object
- **na_i** (`float`) – The illumination NA of the objective.
- **na_c** (`float`) – The collection NA of the objective.
- **center_lambda** (`float`) – The center wavelength of illumination. This should be the center in K-space. So, the wavelength of the center wavenumber.
- **oil_immersion** (`bool`) – If True then the objective is treated as though it is immersed in RI_glass. If False it is treated as though it is immersed in RI_media.
- **cell_glass_interface** (`bool`) – If True then the cell/glass interface is treated as the reference reflection of the configuration. If False then the cell/media interface is treated as the reference reflection.

Return type matlab.object

Returns A S2D.SystemConfiguration MATLAB object.

pwspy.utility.fileIO

Functions for quickly loading files using parallel processing.

Functions

loadAndProcess(fileFrame[, processorFunc, ...])	DEPRECATED! This over-complicated function should be replaced with usage of processParallel.
processParallel(fileFrame, processorFunc[, ...])	A convenience function to process the rows of a pandas DataFrame in parallel

pwspy.utility.fileIO.loadAndProcess

pwspy.utility.fileIO.**loadAndProcess**(*fileFrame*, *processorFunc=None*, *parallel=None*, *procArgs=None*, *initializer=None*, *initArgs=None*)

DEPRECATED! This over-complicated function should be replaced with usage of processParallel. A convenient function to load a series of Data Cubes from a list or dictionary of file paths.

Parameters

- **fileFrame** – A dataframe containing a column of PwsCube file paths titled 'cube' and other columns to act as specifiers for each cube. If no specifiers are used this can just be a list of file paths.
- **processorFunc** – A function that each loaded cell should be passed to. The first argument of processorFunc should be the loaded PwsCube. Additional arguments can be passed to processorFunc using the procArgs variable.
- **parallel** – default is *False*. If *True* then the loading and processing will be performed in parallel on multiple cores, otherwise it will be done using multithreading on a single core. Setting this to true can result if big speedups if the time to run processorFunc is greater than the time to load an PwsCube from file.
- **procArgs** – Optional arguments to pass to processorFunc
- **initializer** – A function that is run once at the beginning of each spawned process. Can be used for copying shared memory.
- **initArgs** – A tuple of arguments to pass to the *initializer* function.

Returns

- *An object of the same form as fileFrame except the PwsCube file paths have been replaced by PwsCube Object.*
- *If using processorFunc the return values from processorFunc will be returned rather than PwsCube Objects.*

pwspy.utility.fileIO.processParallel

pwspy.utility.fileIO.**processParallel**(*fileFrame*, *processorFunc*, *initializer=None*, *initArgs=None*, *procArgs=None*, *numProcesses=None*)

A convenience function to process the rows of a pandas DataFrame in parallel

Parameters

- **fileFrame** (DataFrame) – A dataframe. Each row of the frame will be passed as the first argument to the processorFunc.

- **processorFunc** (Callable[[], Any]) – A function that each row number and row of the *fileFrame* should be passed to as the first and second argument respectively. Additional arguments can be passed to processorFunc using the procArgs variable. The function should return the value which you want included in the return of *processParrallel*.
- **procArgs** (Optional[Tuple]) – Optional arguments to pass to processorFunc
- **initializer** (Optional[Callable]) – A function that is run once at the beginning of each spawned process. Can be used for copying shared memory.
- **initArgs** (Optional[Tuple]) – A tuple of arguments to pass to the *initializer* function.

Return type List containing the results of each execution of *processorFunc*.

pwspy.utility.fluorescence

Functions related to fluorescent images.

Functions

updateFolderStructure(rootDirectory, rotate, ...)	Used to translate old fluorescence images to the new file organization that is recognized by the code.

pwspy.utility.fluorescence.updateFolderStructure

pwspy.utility.fluorescence.**updateFolderStructure**(*rootDirectory*, *rotate*, *flipX*, *flipY*)

Used to translate old fluorescence images to the new file organization that is recognized by the code.

Parameters

- **rootDirectory** (str) – The top level directory containing fluorescence images that were saved in the old *FL_Cell{X}* folder format
- **rotate** (int) – The number of times that the images should be rotated clockwise to match up with the PWS images they go with
- **flipX** (bool) – Should the images be mirrored over the X-axis after being rotated?
- **flipY** (bool) – Should the images be mirrored over the Y-axis after being rotated?

pwspy.utility.machineVision

Useful functions for processing images. Currently its contents are focused on image stabilization.

Functions

to8bit(arr)	Converts boolean or float type numpy arrays to 8bit and scales the data to span from 0 to 255.
SIFTRegisterTransform(reference, other[, ...])	Given a 2D reference image and a list of other images of the same scene but shifted a bit this function will use OpenCV to calculate the transform from each of the other images to the reference.
ORBRegisterTransform(reference, other[, ...])	Given a 2D reference image and a list of other images of the same scene but shifted a bit this function will use OpenCV to calculate the transform from each of the other images to the reference.
edgeDetectRegisterTranslation(reference, other)	This function is used to find the relative translation between a reference image and a list of other similar images.

pwspy.utility.machineVision.to8bit

pwspy.utility.machineVision.**to8bit**(*arr*)

Converts boolean or float type numpy arrays to 8bit and scales the data to span from 0 to 255. Used for many OpenCV functions.

Parameters **arr** (ndarray) – The input array

Return type ndarray

Returns The output array of dtype numpy.uint8

pwspy.utility.machineVision.SIFTRegisterTransform

pwspy.utility.machineVision.**SIFTRegisterTransform**(*reference*, *other*, *mask=None*, *debugPlots=False*)

Given a 2D reference image and a list of other images of the same scene but shifted a bit this function will use OpenCV to calculate the transform from each of the other images to the reference. The transforms can be inverted using cv2.invertAffineTransform(). It will return a list of transforms. Each transform is a 2x3 array in the form returned by opencv.estimateAffinePartial2d(). a boolean mask can be used to select which areas will be searched for features to be used in calculating the transform. This seems to work much better for normalized images. This code is basically a copy of this example, it can probably be improved upon: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html

Parameters

- **reference** (*np.ndarray*) – The 2d reference image.
- **other** (*Iterable[np.ndarray]*) – An iterable containing the images that you want to calculate the translations for.
- **mask** (*np.ndarray*) – A boolean array indicating which parts of the reference image should be analyzed. If *None* then the whole image will be used.

- **debugPlots** (*bool*) – Indicates if extra plots should be openend showing the process of the function.

Returns

A tuple containing: List[np.ndarray]: Returns a list of transforms. Each transform is a 2x3 array in the form returned by opencv.estimateAffinePartial2d(). Note that even though they are returned as affine transforms they will only contain translation information, no scaling, shear, or rotation.

ArtistAnimation: A reference the animation used to diplay the results of the function.

Return type tuple

pwspy.utility.machineVision.ORBRegisterTransform

pwspy.utility.machineVision.**ORBRegisterTransform**(*reference*, *other*, *mask=None*, *debugPlots=False*)

Given a 2D reference image and a list of other images of the same scene but shifted a bit this function will use OpenCV to calculate the transform from each of the other images to the reference. The transforms can be inverted using cv2.invertAffineTransform(). It will return a list of transforms. Each transform is a 2x3 array in the form returned by opencv.estimateAffinePartial2d(). a boolean mask can be used to select which areas will be searched for features to be used in calculating the transform.

Parameters

- **reference** (*np.ndarray*) – The 2d reference image.
- **other** (*Iterable[np.ndarray]*) – An iterable containing the images that you want to calculate the translations for.
- **mask** (*np.ndarray*) – A boolean array indicating which parts of the reference image should be analyzed. If *None* then the whole image will be used.
- **debugPlots** (*bool*) – Indicates if extra plots should be openend showing the process of the function.

Returns

A tuple containing: List[np.ndarray]: Returns a list of transforms. Each transform is a 2x3 array in the form returned by opencv.estimateAffinePartial2d(). Note that even though they are returned as affine transforms they will only contain translation information, no scaling, shear, or rotation.

ArtistAnimation: A reference the animation used to diplay the results of the function.

Return type tuple

pwspy.utility.machineVision.edgeDetectRegisterTranslation

pwspy.utility.machineVision.**edgeDetectRegisterTranslation**(*reference*, *other*, *mask=None*, *debugPlots=False*, *sigma=3*)

This function is used to find the relative translation between a reference image and a list of other similar images. Unlike *SIFRegisterTransforms* this function will not work for images that are rotated relative to the reference. However, it does provide more robust performance for images that do not look identical.

Parameters

- **reference** (*np.ndarray*) – The 2d reference image.

- **other** (*Iterable[np.ndarray]*) – An iterable containing the images that you want to calculate the translations for.
- **mask** (*np.ndarray*) – A boolean array indicating which parts of the reference image should be analyzed. If *None* then the whole image will be used.
- **debugPlots** (*bool*) – Indicates if extra plots should be openend showing the process of the function.
- **sigma** (*float*) – this parameter is passed to skimage.feature.canny to detect edges.

Returns

A tuple containing: list[np.ndarray]: Returns a list of transforms. Each transform is a 2x3 array in the form returned by opencv.estimateAffinePartial2d(). Note that even though they are returned as affine transforms they will only contain translation information, no scaling, shear, or rotation.

list: A list of references to the plotting widgets used to display the results of the function.

Return type tuple

pwspy.utility.micromanager

Objects useful for dealing with files saved by Micro-Manager. https://micro-manager.org/

Classes

Image(directory)	Represents a multi-file Tiff image saved by Micro-Manager
Position1d(z[, stageName, numAxes])	A 1D position usually describing the position of a Z-axis translation stage.
Position2d(x, y[, stageName, numAxes])	Represents a 2D position for a single xy stage in micro-manager.
PositionList(positions)	Represents a micromanager positionList.
Property(value[, pType])	Represents a single property from a micromanager PropertyMap
PropertyMap(properties)	Represents a propertyMap from micromanager.
MultiStagePosition(label, defaultXYStage, ...)	Mirrors the class of the same name from Micro-Manager.

pwspy.utility.micromanager.Image

class pwspy.utility.micromanager.**Image**(*directory*)

Bases: object

Represents a multi-file Tiff image saved by Micro-Manager

Parameters **directory** (str) – The file path to the folder containing the Micro-Manager TIFF files.

pwspy.utility.micromanager.Position1d

class pwspy.utility.micromanager.**Position1d**(*z*, *stageName=''*, *numAxes=1*)
Bases: object

A 1D position usually describing the position of a Z-axis translation stage.

z
The position

Type float

zStage
Then name of the translation stage

pwspy.utility.micromanager.Position2d

class pwspy.utility.micromanager.**Position2d**(*x*, *y*, *stageName=''*, *numAxes=2*)
Bases: object

Represents a 2D position for a single xy stage in micromanager.

x
The x position

Type float

y
The y position

Type float

xyStage
The name of the 2 dimensional translation stage

pwspy.utility.micromanager.PositionList

class pwspy.utility.micromanager.**PositionList**(*positions*)
Bases: object

Represents a micromanager positionList. can be loaded from and saved to a micromanager .pos file.

Parameters **positions** (List[*MultiStagePosition*]) – A list of *MultiStagePosition* objects

positions
A list of *MultiStagePosition* objects

applyAffineTransform(*t*)
Given an affine transformation array this method will transform all positions in this position list. :type t: ndarray :param t: A 2x3 array representing the partial affine transform (rotation, scaling, and translation, but no skew) :type t: np.ndarray

classmethod fromNanoMatFile(*path*, *xyStageName*)
Load an instance of the *PositionList* from a file saved by NC MATLAB acquisition software.

Parameters

- **path** (str) – The file path to the .mat file.
- **xyStageName** (str) – To adapt the MATLAB file format to the Micro-Manager we need to manually supply a name for the

- **stage** (*XY*) –

Returns A new instance of *PositionList*

static fromPropertyMap(*pmap*)
Attempt to load a PositionList from a PropertyMap. May throw an exception.

Return type *PositionList*

getAffineTransform(*otherList*)
Calculate the partial affine transformation between this position list and another position list. Both position lists must have the same length

Parameters **otherList** (PositionList) – A position list of the same length as this position list. Each position is assumed to correspond to the position of the same index in this list.

Returns A 2x3 array representing the partial affine transform (rotation, scaling, and translation, but no skew)

Return type np.ndarray

Examples

a = PositionList.fromNanoMatFile(r'F:/Data/AirDryingSystemComparison/NanoPreDry/corners/positions.mat', "TIXYDRIVE") b = PositionList.fromNanoMatFile(r'F:/Data/AirDryingSystemComparison/NanoPostDry/corners/positions
t = a.getAffineTransform(b) origPos = PositionList.fromNanoMatFile(r'F:/Data/AirDryingSystemComparison/NanoPreDry/0
newPos = origPos.applyAffineTransform(t)

mirrorX()
Invert all x coordinates

Return type *PositionList*

Returns A reference to this object.

mirrorY()
Invert all y coordinates

Return type *PositionList*

Returns A reference to this object.

plot(*fig*, *ax*)
Open a matplotlib plot showing the positions contained in this list.

renameStage(*label*)
Change the name of the xy stage.

Parameters **label** – The new name for the xy Stage

Return type *PositionList*

Returns A reference to this object

toNanoMatFile(*path*)
Save this object to a .mat file in the format saved by NC MATLAB acquistion software.

Parameters **path** (str) – The file path for the new .mat file.

toPropertyMap()
Returns the position list as a PropertyMap that is formatted just like a *PropertyMap* from Micro-Manager.

Return type *PropertyMap*

pwspy.utility.micromanager.Property

class pwspy.utility.micromanager.**Property**(*value*, *pType=None*)

Bases: pwspy.utility.micromanager.PropertyMap._JsonAble

Represents a single property from a micromanager PropertyMap

pType

The type of the property. may be 'STRING', 'DOUBLE', or 'INTEGER'

Type str

value

The value of the propoerty. Should match the type given in *pType*

Type Union[str, int, float, List[Union[str, int, float]]]

encode()

Convert this object to a PropertyMap dictionary.

Return type dict

static hook(*d*)

Check if a dictionary represents an instance of this class and return a new instance. If this dict does not match the correct pattern then just return the original dict.

pwspy.utility.micromanager.PropertyMap

class pwspy.utility.micromanager.**PropertyMap**(*properties*)

Bases: pwspy.utility.micromanager.PropertyMap._JsonAble

Represents a propertyMap from micromanager. basically a list of properties.

properties

A list of properties

encode()

This method should convert the property map class to a dictionary for jsonization

Return type dict

static hook(*d*)

This function should try to identify if the provided JSON object (int, float, string, list, dict) represents an instance of this Property map class. If so then generate the class, otherwire return the input value unchanged.

pwspy.utility.micromanager.MultiStagePosition

class pwspy.utility.micromanager.**MultiStagePosition**(*label*, *defaultXYStage*, *defaultZStage*, *stagePositions*, *gridRow=0*, *gridCol=0*)

Bases: object

Mirrors the class of the same name from Micro-Manager. Can contain multiple Positon1d or Position2d objects. Ideal for a system with multiple translation stages. It is assumed that there is only a single 2D stage and a single 1D stage.

label

A name for the position

Type str

defaultXYStage
The name of the 2D stage

Type str

defaultZStage
The name of the 1D stage

Type str

stagePositions
A list of *Position1d* and *Position2D* objects, usually just one of each.

Type List[Union[*pwspy.utility.micromanager.positions.Position1d*, *pwspy.utility.micromanager.positions.Position2d*]]

copy()
Creates a copy fo the object

Return type *MultiStagePosition*

Returns A new *MultiStagePosition* object.

getXYPosition()
Return the first *Position2d* saved in the *positions* list

getZPosition()
Return the first *Position1d* saved in the positions` list. Returns *None* if no position is found.

renameXYStage(*label*)
Change the name of the xy stage.

Parameters **label** (str) – The new name for the xy Stage

pwspy.utility.misc

Objects that are generally useful in python programming.

Decorators

cached_property(func)	A decorator for a property that is only computed once per instance and then replaces itself with an ordinary attribute.
profileDec(filePath)	A decorator to profile a function call using cProfile

pwspy.utility.misc.cached_property

class pwspy.utility.misc.**cached_property**(*func*)
Bases: object

A decorator for a property that is only computed once per instance and then replaces itself with an ordinary attribute. Deleting the attribute resets the property. Source: https://github.com/bottlepy/bottle/commit/fa7733e075da0d790d809aa3d2f53071897e6f76

pwspy.utility.misc.profileDec

pwspy.utility.misc.**profileDec**(*filePath*)
A decorator to profile a function call using cProfile

Parameters **filePath** (str) – cProfile will dump a log file to this location.

pwspy.utility.plotting

Image Plotting

Functions

roiColor(data, rois, vmin, vmax, scale_bg[, ...])	Given a 2D image of data this function will scale the data, apply an exponential curve, and color the ROI regions with Hue.

pwspy.utility.plotting.roiColor

pwspy.utility.plotting.**roiColor**(*data*, *rois*, *vmin*, *vmax*, *scale_bg*, *hue=0*, *exponent=1*, *numScaleBarPix=0*)
Given a 2D image of data this function will scale the data, apply an exponential curve, and color the ROI regions with Hue. Used in many presentations and publications.

Parameters

- **data** (*np.ndarray*) – an MxN array of data to be imaged
- **rois** (*List[Roi]*) – a list of Roi objects. the regions inside a roiFile will be colored.
- **vmin** (*float*) – the minimum value in the data that will be set to black
- **vmax** (*float*) – the maximum value in the data that will be set to white
- **scale_bg** (*float*) – Scales the brightness of the background (non-roiFile) region.
- **hue** (*float*) – A value of 0-1 indicating the hue of the colored regions.
- **exponent** (*float*) – The exponent used to curve the color map for more pleasing results.
- **numScaleBarPix** (*float*) – The length of the scale bar in number of pixels.

Returns MxNx3 RGB array of the image

Return type np.ndarray

pwspy.utility.reflection

A package containing functionality useful for calculation reflections.

Subpackages

extraReflectance	A collection of functions dedicated to the purpose of generating Extra Reflectance calibrations from images of materials with known reflectances (e.g.
multilayerReflectanceEngine	Using the wave transfer matrix formalism from chapter 7 of Saleh and Teich Fundamentals of Photonics, this script calculates the reflectance of a multilayer dielectric.
reflectanceHelper	Provides a number of functions for calculating simple reflections based on known refractive indices

pwspy.utility.reflection.extraReflectance

A collection of functions dedicated to the purpose of generating Extra Reflectance calibrations from images of materials with known reflectances (e.g. air/glass interface, water/glass interface.)

By calculating the "extra reflectance" of a microscope system we can come up with a subtraction from our raw data that will make our ratiometric measurements proportional to the actual sample reflectance.

These functions are relied on heavily in "ERCreator" app found in *pwspy_gui.ExtraReflectanceCreator*.

Functions

getTheoreticalReflectances(materials, ...)	Generate a dictionary containing a Pandas *Series* of the *material*-glass reflectance for each material in *materials*.
generateMaterialCombos(materials[, ...])	Given a list of materials, this function returns a list of all possible material combo tuples.
getAllCubeCombos(matCombos, cubeDict)	Given a list of material combo tuples, return a dictionary whose keys are the material combo tuples and whose values are lists of CubeCombos.
plotExtraReflection(images, theoryR, matCombos)	Generate a variety of plots displaying information about the extra reflectance calculation.
generateRExtraCubes(allCombos, theoryR, ...)	Generate a series of extra reflectance cubes based on the input data.

pwspy.utility.reflection.extraReflectance.getTheoreticalReflectances

pwspy.utility.reflection.extraReflectance.**getTheoreticalReflectances**(*materials*, *wavelengths*, *numericalAperture*)

Generate a dictionary containing a Pandas *Series* of the *material*-glass reflectance for each material in *materials*.

Parameters

- **materials** (Set[*Material*]) – The set of materials that you want to retrieve the theoretical reflectance for.
- **wavelengths** (Tuple[float]) – The wavelengths that you want the reflectances calculated at.
- **numericalAperture** (float) – The numerical aperture that the reflectance should be calculated at.

Return type Dict[*Material*, Series]

Returns A dictionary of the reflectances for each material. The material serves as the dictionary key.

pwspy.utility.reflection.extraReflectance.generateMaterialCombos

pwspy.utility.reflection.extraReflectance.**generateMaterialCombos**(*materials*, *excludedCombos=None*)

Given a list of materials, this function returns a list of all possible material combo tuples.

Parameters

- **materials** (Iterable[*Material*]) – The list of materials that you want to generate every possible combo of.
- **excludedCombos** (Optional[Iterable[Tuple[*Material*, *Material*]]]) – Combinations of materials that you don't want included in the combinations.

Return type List[Tuple[*Material*, *Material*]]

Returns A list of Material combinations.

pwspy.utility.reflection.extraReflectance.getAllCubeCombos

pwspy.utility.reflection.extraReflectance.**getAllCubeCombos**(*matCombos*, *cubeDict*)

Given a list of material combo tuples, return a dictionary whose keys are the material combo tuples and whose values are lists of CubeCombos.

Parameters

- **matCombos** (Iterable[Tuple[*Material*, *Material*]]) – A list of material combinations, most likely generated by *generateMaterialCombos*
- **cubeDict** (Dict[*Material*, List[*PwsCube*]]) – An dictionary containing lists of a PwsCube measuremnts keyed by the material they were measured at.

Return type Dict[Tuple[*Material*, *Material*], List[CubeCombo]]

Returns A dictionary with a key for each material combination. Each value is a list of all the *CubeCombo`s extracted from `cubes*.

pwspy.utility.reflection.extraReflectance.plotExtraReflection

pwspy.utility.reflection.extraReflectance.**plotExtraReflection**(*images*, *theoryR*, *matCombos*, *mask=None*)

Generate a variety of plots displaying information about the extra reflectance calculation.

Parameters

- **images** (Dict[str, Dict[*Material*, List[*PwsCube*]]]) – A dictionary where the keys are strings representing some configuration of the system and the values are dictionaries where the keys are a *Material* and the values are lists of the *PwsCube* that were measured at the corresponding glass-{material} interface and configuration indicated by the dictionary keys.
- **theoryR** (Dict[*Material*, Series]) – A dictionary where the key is a *Material* and the value is a Pandas 'Series' giving the reflectance for a glass-{material} reflection over a range of wavelengths. The index of the series should be the wavelengths.

- **matCombos** (List[Tuple[*Material*, *Material*]]) – A list of the various material combinations that should be evaluated.
- **mask** (Optional[*Roi*]) – An ROI indicating the region of the images that should be included in the evaluation.

Return type List[Figure]

Returns A list of matplotlib figures resulting from this calculation.

pwspy.utility.reflection.extraReflectance.generateRExtraCubes

pwspy.utility.reflection.extraReflectance.**generateRExtraCubes**(*allCombos*, *theoryR*, *numericalAperture*)

Generate a series of extra reflectance cubes based on the input data.

Parameters

- **allCombos** (Dict[Tuple[*Material*, *Material*], List[CubeCombo]]) – a dict of lists CubeCombos, each keyed by a 2-tuple of Materials.
- **theoryR** (Dict[*Material*, Series]) – the theoretically predicted reflectance for each material.
- **numericalAperture** (float) – The numerical aperture that the PwsCubes were imaged at. The theoryR reflectances should have also been calculated at this NA

Return type Tuple[*ExtraReflectanceCube*, Dict[Union[str, Tuple[*Material*, *Material*]], ndarray]]

Returns

An *ExtraReflectanceCube* object containing data from the weighted average of all measurements. A dictionary where the keys are material combos and the values are tuples of the weighted-Mean and the weight arrays.

Classes

pwspy.utility.reflection.multilayerReflectanceEngine

Using the wave transfer matrix formalism from chapter 7 of Saleh and Teich Fundamentals of Photonics, this script calculates the reflectance of a multilayer dielectric. http://www.phys.ubbcluj.ro/~emil.vinteler/nanofotonica/TemeControl_FCMD014_Vinteler.pdf https://en.wikipedia.org/wiki/Transfer-matrix_method_(optics)

m is the final transfer matrix. It should be made by multiplying the matrices representing each element of the system. If the transmitted light is considered to be propagating from left to right then the matrices should be in multiplied in reverse, from right to left.

Classes

Polarization(value)	An enumeration of the possible polarization types.
Layer(mat, d[, name])	This represents a layer with a thickness and an index of refraction.
Stack(wavelengths[, elements])	Represents a stack of 1d homogenous films.
NonPolarizedStack(wavelengths[, elements])	Represents a stack of 1d homogenous films.

pwspy.utility.reflection.multilayerReflectanceEngine.Polarization

class pwspy.utility.reflection.multilayerReflectanceEngine.**Polarization**(*value*)
Bases: enum.Enum

An enumeration of the possible polarization types.

pwspy.utility.reflection.multilayerReflectanceEngine.Layer

class pwspy.utility.reflection.multilayerReflectanceEngine.**Layer**(*mat*, *d*, *name=None*)
Bases: object

This represents a layer with a thickness and an index of refraction. Note: This whole system only supports lossless media, we only use the real part of the index of refraction.

Parameters

- **mat** (Union[Number, Series, *Material*]) – This can either be a number or series of numbers representing the refractive index at different wavelengths or it can be a *Material* in which case the refractive index will be automatically calculated.
- **d** (float) – The thickness of the layer. The units that thicknesses and wavelengths are specified in must match.
- **name** (Optional[str]) – An optional name which will be dislayed if the layer is plotted

getRefractiveIndex(*wavelengths*)
Get the refractive index of the layer.

Parameters **wavelengths** (ndarray) – The wavelengths to calculate the refractive index at.

Return type Series

Returns The refractive index.

pwspy.utility.reflection.multilayerReflectanceEngine.Stack

class pwspy.utility.reflection.multilayerReflectanceEngine.**Stack**(*wavelengths*, *elements=None*)
Bases: pwspy.utility.reflection.multilayerReflectanceEngine.StackBase

Represents a stack of 1d homogenous films. Reflectance for the two polarizations can be calculated for a range of numerical apertures (angles). Indices of refraction must be real (no absorption).

Parameters

- **wavelengths** (Union[Number, ndarray]) – The wavelengths that calculation should operate over.

- **elements** (Optional[List[*Layer*]]) – The initial layers to add to the stack.

addLayer(*element*)
Add a new *Layer* to the *Stack*

Parameters **element** (*Layer*) – A new layer to add.

calculateReflectance(*NAs*)
Given an array of numerical apertures this function returns the reflectance as a dictionary of 2d arrays. There is one 2d array for each of the two polarizations. the dimensions of the array is (wavelengths x NAs). The total reflectance can be calculated as the average reflectance of the two polarizations. Other ellipsometric parameters can also be calculated.

Parameters **NAs** (ndarray) – The numerical apertures to calculate reflectance at.

Returns A dictionary containing a reflectance array for each of the two polarizations. The polarization is the key to the dictionary. Each reflectance is a MxN array where M is the number of wavelengths and N is the number of NAs passed to this function.

circularIntegration(*NAs*)
Given an array of NumericalApertures (usually from 0 to NAMax.) This function integrates the reflectance over a disc of Numerical Apertures (Just like in a microscope the Aperture plane is a disc shape, with higher NA being further from the center.) Ultimately the result of this integration should match the reflectance measured with the same NA.

Parameters **NAs** (ndarray) – The numerical apertures to calculate reflectance at.

Return type Series

Returns A pandas *Series* with wavelengths as the index and reflectance as the value.

static interfaceMatrix(*n1*, *n2*, *polarization*, *NAs*)
Returns a matrix representing a dieletric interface. n1 and n2 should be a pandas Series where the values are complex refractive index and the index of the Series is the associated wavelengths.

Parameters

- **n1** (Series) – The refractive indices on one side of the reflective interface
- **n2** (Series) – The refractive indices on the other side of the reflective interface
- **polarization** (*Polarization*) – The polarization that should be used for the calculation
- **NAs** (ndarray) – An array of the numerical aperture values. :todo: More details would be good

Return type ndarray

Returns A transfer matrix for the reflective interface

plot(*NAs*, *polarization=None*)
Plot various graphs of reflectance vs NA. NAs should be an array of Numerical apertures to have the reflectance calculated for. *polarization* can be specified to view the reflectance of only one polarization.

static propagationMatrix(*n*, *d*, *NAs*)
Returns a matrix representing the propagation of light for a distance of *d*. d and the wavelengths must use the same units.

Parameters

- **n** (Series) – The refractive indices of the material
- **d** (float) – The distance of propagation.

Return type ndarray

Returns A transfer matrix for propagation through a material.

pwspy.utility.reflection.multilayerReflectanceEngine.NonPolarizedStack

class pwspy.utility.reflection.multilayerReflectanceEngine.**NonPolarizedStack**(*wavelengths*, *elements=None*)

Bases: pwspy.utility.reflection.multilayerReflectanceEngine.StackBase

Represents a stack of 1d homogenous films. Reflectance can only be calculated at 0 incidence angle in which case polarization is irrelevant. This class does not do anything that can't be done with *Stack*. Indices of refraction must be real (no absorption).

Parameters

- **wavelengths** (Union[Number, ndarray]) – The wavelengths that calculation should operate over.
- **elements** (Optional[List[*Layer*]]) – The initial layers to add to the stack.

addLayer(*element*)

Add a new *Layer* to the *Stack*

Parameters **element** (*Layer*) – A new layer to add.

calculateReflectance()

Calculate the reflectance for this *Stack*.

Return type ndarray

Returns The reflectance.

static interfaceMatrix(*n1*, *n2*)

Generate a matrix representing the interface between two dielectrics with indices n1 on the left and n2 on the right. Actually the order of terms does not appear to matter. This does not account for polarization or incidence angles other than 0 degrees.

Parameters

- **n1** (Series) – The refractive indices on one side of the reflective interface
- **n2** (Series) – The refractive indices on the other side of the reflective interface

Return type ndarray

Returns A transfer matrix for the reflective interface

plot()

Open a Matplotlib plot of the stack.

static propagationMatrix(*n*, *d*)

Returns a matrix representing the propagation of light. n should be a pandas Series where the values are complex refractive index and the index fo the Series is the associated wavelengths. with wavelength. for a distance of *d*. d and the wavelengths must use the same units.

Parameters

- **n** (Series) – The refractive indices of the material
- **d** (float) – The distance of propagation.

Return type ndarray

Returns A transfer matrix for propagation through a material.

pwspy.utility.reflection.reflectanceHelper

Provides a number of functions for calculating simple reflections based on known refractive indices

Functions

getReflectance(mat1, mat2[, wavelengths, NA])	Given the names of two interfaces this provides the reflectance in units of percent.
getRefractiveIndex(mat[, wavelengths])	Get the spectrally dependent refractive index of a material.

pwspy.utility.reflection.reflectanceHelper.getReflectance

pwspy.utility.reflection.reflectanceHelper.**getReflectance**(*mat1*, *mat2*, *wavelengths=None*, *NA=0*)

Given the names of two interfaces this provides the reflectance in units of percent. If given a series as wavelengths the data will be interpolated and reindexed to match the wavelengths.

Parameters

- **mat1** (Union[Number, Series, *Material*]) – The first material comprising the reflective interface. This can either be a number or series of numbers representing the refractive index at different wavelengths or it can be a *Material* in which case the refractive index will be automatically calculated.
- **mat2** (Union[Number, Series, *Material*]) – The second material comprising the reflective interface. This can either be a number or series of numbers representing the refractive index at different wavelengths or it can be a *Material* in which case the refractive index will be automatically calculated.
- **wavelengths** (Union[ndarray, List, Tuple, None]) – The wavelengths to calculate the reflectance at.
- **NA** (float) – The numerical aperture of the system. Reflectance will be calculated by radially integrating results over the range of angles present within the numerical aperture. If left as *None* the result is calculated for light with a 0 degree angle of incidence.

Return type Series

Returns The percentage reflectance. The index of the pandas Series is the wavelengths.

pwspy.utility.reflection.reflectanceHelper.getRefractiveIndex

pwspy.utility.reflection.reflectanceHelper.**getRefractiveIndex**(*mat*, *wavelengths=None*)

Get the spectrally dependent refractive index of a material.

Parameters

- **mat** (*Material*) – The material the retrieve the refractive index of.
- **wavelengths** (Optional[Iterable[float]]) – The wavelengths that the refractive index should be calculated at. If left as *None* then the wavelengths used will be determined by the original file that the data was pulled from.

Return type Series

Returns The refractive index. The index of the pandas series is the wavelengths.

Classes

Material(value)	An enumeration class containing items for the various materials that we can calculate reflectance for.

pwspy.utility.reflection.Material

class pwspy.utility.reflection.**Material**(*value*)

Bases: enum.Enum

An enumeration class containing items for the various materials that we can calculate reflectance for.

CHAPTER

THREE

EXAMPLES

3.1 Examples

If you don't have PWS experimental data to run these examples with you can find a dataset that is used for automated testing on Zenodo here: https://zenodo.org/record/5976039#.Yf6aPt_MJPY

3.1.1 Running basic PWS analysis on a single image.

This example runs the PWS analysis on a single measurement. The results of the analysis will be saved alongside the raw data under the *analyses* folder. Running this example may be required in order for other examples which require that analysis results already be available.

```
# -*- coding: utf-8 -*-
# Copyright 2018-2021 Nick Anthony, Backman Biophotonics Lab, Northwestern University
#
# This file is part of PWSpy.
#
# PWSpy is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# PWSpy is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with PWSpy.  If not, see <https://www.gnu.org/licenses/>.

"""
This script saves pws analysis results to the test data. This must be run before many of␣
↪the other examples will work.
"""

from pwspy import analysis
from pwspy import dataTypes as pwsdt
import pathlib as pl

### User Variables ###
```

(continues on next page)

(continued from previous page)

```
PWSExperimentPath: pl.Path = ...  # Set this to a folder containing multiple "Cell{x}" acquisition folders. If you have downloaded the test dataset you can use `pl.Path(__file__).parent.parent / 'tests' / 'resources' / 'test_data' / 'sequencer'`
######################

settings = analysis.pws.PWSAnalysisSettings.loadDefaultSettings("Recommended")

# Load our blank reference image which will be used for normalization.
refAcq = pwsdt.Acquisition(PWSExperimentPath / 'Cell3')
ref = refAcq.pws.toDataClass()

anls = analysis.pws.PWSAnalysis(settings=settings, extraReflectance=None, ref=ref)  # Create a new analysis for the given reference image and analysis settings. The "ExtraReflection" calibration is ignored in this case.

acq = pwsdt.Acquisition(PWSExperimentPath / "Cell1")  # Create an "Acquisition" object to handle operations for the data associated with a single acquisition
cube = acq.pws.toDataClass()  # Request that the PWS metadata object load the full data.
results, warnings = anls.run(cube)  # Run the pre-setup analysis on our data. Get the analysis results and potentially a list of warnings from the analysis.

acq.pws.saveAnalysis(results, 'myAnalysis', overwrite=True)  # Save our analysis results to file in the default location alongside the raw data under the `analyses` folder.

loadedResults = acq.pws.loadAnalysis('myAnalysis')  # This is just going to be a copy of `results`, loaded from file.

print(f"Saved anlysis `myAnalysis` to {acq.filePath}")
```

3.1.2 Performing FFT to visualize the estimated depth of cell features

```
# -*- coding: utf-8 -*-
# Copyright 2018-2021 Nick Anthony, Backman Biophotonics Lab, Northwestern University
#
# This file is part of PWSpy.
#
# PWSpy is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# PWSpy is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with PWSpy.  If not, see <https://www.gnu.org/licenses/>.

"""
```

(continues on next page)

(continued from previous page)

```
Load the OPD from a previously saved analysis result and plot it using a special multi-dimensional plotting widget.

@author: Nick Anthony
"""

import pwspy.dataTypes as pwsdt
from mpl_qt_viz.visualizers import PlotNd  # The mpl_qt_viz library is not a dependency of PWSpy and will need to be installed separately. It can be found on Conda-Forge (conda install -c conda-forge mpl_qt_viz) or on PyPi (pip install mpl_qt_viz)
import matplotlib.pyplot as plt
import pathlib as pl
from PyQt5.QtWidgets import QApplication

### User Variables ###
PWSImagePath: pl.Path = ... # Set this to the path of a "Cell{X}" folder of a PWS acquisition.
######################

plt.ion()  # Without this we will get a crash when trying to open the PlotNd window because a Qt application loop must be running.
plt.figure()

acquisition = pwsdt.Acquisition(PWSImagePath)  # Get a reference to the top-level folder for the measurement.

roiSpecs = acquisition.getRois()  # Get a list of the (name, number, file format) of the available ROIs.
print("ROIs:\n", roiSpecs)

analysis = acquisition.pws.loadAnalysis(acquisition.pws.getAnalyses()[0])  # Load a reference to an analysis file.
kCube = analysis.reflectance  # Load the processed `reflectance` array from the analysis file.

opd, opdValues = kCube.getOpd(useHannWindow=False, indexOpdStop=50)  # Use FFT to transform the reflectance array to OPD

# Scale the opdValues to give estimated depth instead of raw OPD. Factor of 2 because light is making a round-trip.
ri = 1.37  # Estimated RI of livecell chromatin
opdValues = opdValues / (2 * ri)

app = QApplication([])  # Start a QApplication instance to run the PlotNd qwidget
plotWindow = PlotNd(opd, names=('y', 'x', 'depth'),
                    indices=(None, None, opdValues), title="Estimated Depth")
app.exec()
```

3.1.3 Compile analysis results to a table of average values within each ROI

```python
# Copyright 2018-2021 Nick Anthony, Backman Biophotonics Lab, Northwestern University
#
# This file is part of PWSpy.
#
# PWSpy is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# PWSpy is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with PWSpy.  If not, see <https://www.gnu.org/licenses/>.

"""
This script loads analysis results from a list of PWS acquisitions and uses the "PWSRoiCompiler" class to get average output
values within the ROIs for each acquisition. The compiled results are then placed into a Pandas dataframe.
"""
import pandas
from pwspy.analysis.compilation import PWSRoiCompiler, PWSCompilerSettings
import pwspy.dataTypes as pwsdt
import pathlib as pl

### User Variables ###
PWSExperimentPath: pl.Path = ...  # Set this to a folder containing multiple "Cell{x}" acquisition folders. If you have downloaded the test dataset you can use `pl.Path(__file__).parent.parent / 'tests' / 'resources' / 'test_data' / 'sequencer'`
######################


tmpList = []
compiler = PWSRoiCompiler(PWSCompilerSettings(reflectance=True, rms=True))

listOfAcquisitions = [pwsdt.Acquisition(i) for i in PWSExperimentPath.glob("Cell[0-9]")]
for acquisition in listOfAcquisitions:
    for analysisName in acquisition.pws.getAnalyses():
        analysisResults = acquisition.pws.loadAnalysis(analysisName)
        for roiSpec in acquisition.getRois():
            roiFile = acquisition.loadRoi(*roiSpec)
            results, warnings = compiler.run(analysisResults, roiFile.getRoi())

            if len(warnings) > 0:
                print(warnings)

            tmpList.append(dict(
                acquisition=acquisition,
```

(continues on next page)

(continued from previous page)

```
                cellNumber=acquisition.getNumber(),
                analysisResults=analysisResults,
                rms=results.rms,
                reflectance=results.reflectance,
                roiNum=roiFile.number,
                roiName=roiFile.name
            ))

dataFrame = pandas.DataFrame(tmpList)
print(dataFrame)
```

3.1.4 Basic loading of ROI's to extract data from specific regions

```
# -*- coding: utf-8 -*-
# Copyright 2018-2021 Nick Anthony, Backman Biophotonics Lab, Northwestern University
#
# This file is part of PWSpy.
#
# PWSpy is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# PWSpy is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with PWSpy.  If not, see <https://www.gnu.org/licenses/>.

"""
Loop through all ROIs for all acquisitions in a directory and plot a histogram of the␣
↪RMS values within the ROI.
"""

import pwspy.dataTypes as pwsdt
import matplotlib.pyplot as plt
import pathlib
import numpy as np


### User Variables ###
PWSExperimentPath = ...  # Set this to a folder containing multiple "Cell{x}"␣
↪acquisition folders. If you have downloaded the test dataset you can use `pl.Path(__
↪file__).parent.parent / 'tests' / 'resources' / 'test_data' / 'sequencer'`
analysisName = 'script'  # This will often be "p0"
######################

plt.ion()
```

(continues on next page)

(continued from previous page)

```


def plotHist(roi, rms):
    """
    This function takes an ROI
    and a 2D RMS image and plots a histogram of the RMS values inside the ROI
    """
    # Check input values just to be safe.
    assert isinstance(roi, pwsdt.Roi)  # Make sure roiFile variable is actually an ROI
    assert isinstance(rms, np.ndarray)  # Make sure the RMS image is an numpy array
    assert roi.mask.shape == rms.shape  # Make sure the ROI and RMS arrays have the same␣
↪dimensions.

    vals = rms[roi.mask]  # A 1D array of the values inside the ROI
    plt.hist(vals)  # Plot a histogram


cellFolderIterator = pathlib.Path(PWSExperimentPath).glob("Cell[0-9]")  # An iterator␣
↪for all folders that are below workingDirectory and match the "regex" pattern "Cell[0-
↪9]"
for folder in cellFolderIterator:
    acq = pwsdt.Acquisition(folder)  # An object handling the contents of a single "Cell
↪{X}" folder

    try:
        anls = acq.pws.loadAnalysis(analysisName)  # Load the analysis results from file.
    except:
        print(f"Analysis loading failed for {acq.filePath}")
        continue  # Skip to the next loop iteration

    roiSpecs = acq.getRois()  # A list of the names, numbers, and fileFormats of the␣
↪ROIs in this acquisition

    for name, number, fformat in roiSpecs:  # Loop through every ROI.
        roiFile = acq.loadRoi(name, number, fformat)  # Load the ROI from file.
        plotHist(roiFile.getRoi(), anls.rms)  # Use the function defined above to plot a␣
↪histogram
```

3.1.5 Using a hand-drawn ROI to generate a reference pseudo-measurement

```

# -*- coding: utf-8 -*-
# Copyright 2018-2021 Nick Anthony, Backman Biophotonics Lab, Northwestern University
#
# This file is part of PWSpy.
#
# PWSpy is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
```

(continues on next page)

(continued from previous page)

```
#
# PWSpy is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with PWSpy.  If not, see <https://www.gnu.org/licenses/>.

"""
This script allows the user to select a region of an PwsCube. the spectra of this
region is then averaged over the X and Y dimensions. This spectra is then saved
as a reference dataTypes with the same initial dimensions.
Can help to make a reference when you don't actually have one for some reason
"""

import pwspy.dataTypes as pwsdt
import matplotlib.pyplot as plt
import numpy as np
import pathlib as pl

### User Variables ###
PWSImagePath: pl.Path = ... # Set this to the path of a "Cell{X}" folder of a PWS acquisition.
######################

plt.ion()
a = pwsdt.Acquisition(PWSImagePath).pws.toDataClass()  # Load a measurement from file.

roi = a.selectLassoROI()  # Prompt the user for a hand-drawn ROI
spec, std = a.getMeanSpectra(mask=roi)  # Get the average spectra within the ROI
newData = np.zeros(a.data.shape)
newData[:, :, :] = spec[np.newaxis, np.newaxis, :]  # Extend the averaged spectrum along the full dimensions of the original measurement.
ref = pwsdt.PwsCube(newData, a.metadata)  # Create a new synthetic measurement using the averaged spectrum
plt.plot(a.wavelengths, spec)

```

3.1.6 Blurring data laterally to smooth a reference image.

```
# -*- coding: utf-8 -*-
# Copyright 2018-2021 Nick Anthony, Backman Biophotonics Lab, Northwestern University
#
# This file is part of PWSpy.
#
# PWSpy is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
```

(continues on next page)

(continued from previous page)

```
#
# PWSpy is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with PWSpy.  If not, see <https://www.gnu.org/licenses/>.

"""
This script blurs an image cube in the xy direction. Allows you to turn an
image of cells into something that can be used as a reference image, assuming
most of the the FOV is glass. In reality you should just have a good reference image to␣
↪use and not resort to something
like this.
"""

import copy
import matplotlib.pyplot as plt
import pwspy.dataTypes as pwsdt

### User Variables ###
PWSImagePath = ...  # Set this to the path of a "Cell{X}" folder of a PWS acquisition.
######################

plt.ion()

acq = pwsdt.Acquisition(PWSImagePath)
a = acq.pws.toDataClass()

a.correctCameraEffects()  # Correct for dark counts and potentially for camera␣
↪nonlinearity using metadata stored with the original measurement.
a.normalizeByExposure()  # Divide by exposure time to get data in units of `counts/ms`.␣
↪This isn't strictly necessary in this case.

mirror = copy.deepcopy(a)
mirror.filterDust(10)  # Apply a gaussian blurring with sigma=10 microns along the XY␣
↪plane.

a.plotMean()  # Plot the mean reflectance of the original
mirror.plotMean()  # Plot the mean reflectance after filtering.
a.normalizeByReference(mirror)  # Normalize raw by reference
a.plotMean()  # Plot the measurement after normalization.
plt.figure()
plt.imshow(a.data.std(axis=2))  # Plot RMS after normalization.
```

3.1.7 Measuring Sigma using only a limited range of the OPD signal.

```

# -*- coding: utf-8 -*-
# Copyright 2018-2021 Nick Anthony, Backman Biophotonics Lab, Northwestern University
#
# This file is part of PWSpy.
#
# PWSpy is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# PWSpy is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with PWSpy.  If not, see <https://www.gnu.org/licenses/>.

"""
This script is based on a matlab script written by Lusik Cherkezyan for NC.
Nano uses this method to extract rms from phantom make from ChromEM cells embedded in resin.
The phantom has a strong thin-film spectrum. This script is meant to filter out the thin film components
of the fourier transfrom and extract RMS from what is left.
"""
from pwspy.dataTypes import CameraCorrection, Acquisition, PwsCube, KCube
import matplotlib.pyplot as plt
import scipy.signal as sps
import os
import numpy as np
import pathlib as pl

'''User Input'''
path: pl.Path = ...
refName = 'Cell3'  # This is an PwsCube of glass, used for normalization.
cellNames = ['Cell1', 'Cell2']  # , 'Cell3', 'Cell4','Cell5']
maskSuffix = 'resin'

# identify the depth in um to which the OPD spectra need to be integrated
integrationDepth = 2.0  # in um
isHannWindow = True  # Should Hann windowing be applied to eliminate edge artifacts?
subtractResinOpd = True
resetResinMasks = False
wvStart = 510  # start wavelength for poly subtraction
wvEnd = 690  # end wavelength for poly subtraction
sampleRI = 1.545  # The refractive index of the resin. This is taken from matlab code, I don't know if it's correct.
orderPolyFit = 0
wv_step = 2
```

(continues on next page)

(continued from previous page)

```
correction = CameraCorrection(2000, (0.977241216, 1.73E-06, 1.70E-11))

'''**************'''

b, a = sps.butter(6, 0.1 * wv_step)
opdIntegralEnd = integrationDepth * 2 * sampleRI  # We need to convert from our desired depth into an opd value. There are some questions about having a 2 here but that's how it is in the matlab code so I'm keeping it.

### load and save mirror or glass image cube
ref = PwsCube.fromMetadata(Acquisition(os.path.join(path, refName)).pws)
ref.correctCameraEffects(correction)
ref.filterDust(6, pixelSize=1)
ref.normalizeByExposure()

if subtractResinOpd:
    ### load and save reference empty resin image cube
    fig, ax = plt.subplots()
    resinOpds = {}
    for cellName in cellNames:
        resin = PwsCube.fromMetadata(Acquisition(os.path.join(path, cellName)).pws)
        resin.correctCameraEffects(correction)
        resin.normalizeByExposure()
        resin /= ref
        resin = KCube.fromPwsCube(resin)
        if resetResinMasks:
            [resin.metadata.acquisitionDirectory.deleteRoi(name, num) for name, num, fformat in resin.metadata.acquisitionDirectory.getRois() if name == maskSuffix]
        if maskSuffix in [name for name, number, fformat in resin.metadata.acquisitionDirectory.getRois()]:
            resinRoi = resin.metadata.acquisitionDirectory.loadRoi(maskSuffix, 1).getRoi()
        else:
            print('Select a region containing only resin.')
            resinRoi = resin.selectLassoROI()
            resin.metadata.acquisitionDirectory.saveRoi(maskSuffix, 1, resinRoi)
        resin.data -= resin.data.mean(axis=2)[:, :, np.newaxis]
        opdResin, xvals = resin.getOpd(isHannWindow, indexOpdStop=None, mask=resinRoi.mask)
        resinOpds[cellName] = opdResin
        ax.plot(xvals, opdResin, label=cellName)
        ax.vlines([opdIntegralEnd], ymin=opdResin.min(), ymax=opdResin.max())
        ax.set_xlabel('OPD')
        ax.set_ylabel("Amplitude")
    ax.legend()
    plt.pause(0.2)

print("Beginning processing.")
rmses = {}  # Store the rms maps for later saving
for cellName in cellNames:
    cube = PwsCube.fromMetadata(Acquisition(os.path.join(path, cellName)).pws)
    cube.correctCameraEffects(correction)
```

(continues on next page)

(continued from previous page)

```
    cube.normalizeByExposure()
    cube /= ref
    cube.data = sps.filtfilt(b, a, cube.data, axis=2)
    cube = KCube.fromPwsCube(cube)

    ## -- Polynomial Fit
    print("Subtracting Polynomial")
    polydata = cube.data.reshape((cube.data.shape[0] * cube.data.shape[1], cube.data.
↪shape[2]))
    polydata = np.rollaxis(polydata, 1)  # Flatten the array to 2d and put the␣
↪wavenumber axis first.
    cubePoly = np.zeros(polydata.shape)  # make an empty array to hold the fit values.
    polydata = np.polyfit(cube.wavenumbers, polydata,
                          orderPolyFit)  # At this point polydata goes from holding the␣
↪cube data to holding the polynomial values for each pixel. still 2d.
    for i in range(orderPolyFit + 1):
        cubePoly += (np.array(cube.wavenumbers)[:, np.newaxis] ** i) * polydata[i,
                                                                       :]  # Populate␣
↪cubePoly with the fit values.
    cubePoly = np.moveaxis(cubePoly, 0, 1)
    cubePoly = cubePoly.reshape(cube.data.shape)  # reshape back to a cube.
    # Remove the polynomial fit from filtered cubeCell.
    cube.data = cube.data - cubePoly

    rmsData = np.sqrt(np.mean(cube.data ** 2, axis=2)) #This can be compared to␣
↪rmsOPDIntData, when the integralStopIdx is high and we don't do spectral subtraction␣
↪they should be equivalent.

    # Find the fft for each signal in the desired wavelength range
    opdData, xvals = cube.getOpd(isHannWindow, None)

    if subtractResinOpd:
        opdData = opdData - resinOpds[cellName]

    try:
        integralStopIdx = np.where(xvals >= opdIntegralEnd)[0][0]
    except IndexError:  # If we get an index error here then our opdIntegralEnd is␣
↪probably bigger than we can achieve. Just use the biggest value we have.
        integralStopIdx = None
        opdIntegralEnd = max(xvals)
        print(f'Integrating to OPD {opdIntegralEnd}')

    opdSquared = np.sum(opdData[:, :, :integralStopIdx] ** 2,axis=2)  # Parseval's␣
↪theorem tells us that this is equivalent to the sum of the squares of our original␣
↪signal
    opdSquared *= len(cube.wavenumbers) / opdData.shape[2]  # If the original data and␣
↪opd were of the same length then the above line would be correct. Since the fft has␣
↪been upsampled. we need to normalize.
    rmsOpdIntData = np.sqrt(opdSquared)  # this should be equivalent to normal RMS if␣
↪our stop index is high and resin subtraction is disabled.

    cmap = plt.get_cmap('jet')
```

(continues on next page)

(continued from previous page)

```
    fig, axs = plt.subplots(1, 2, sharex=True, sharey=True)
    im = axs[0].imshow(rmsData, cmap=cmap, clim=[np.percentile(rmsData, 0.5), np.
↪percentile(rmsData, 99.5)])
    fig.colorbar(im, ax=axs[0])
    axs[0].set_title('RMS')
    im = axs[1].imshow(rmsOpdIntData, cmap=cmap,
                       clim=[np.percentile(rmsOpdIntData, 0.5), np.
↪percentile(rmsOpdIntData, 99.5)])
    fig.colorbar(im, ax=axs[1])
    axs[1].set_title(f'RMS from OPD below {opdIntegralEnd} after resin OPD subtraction')
    fig.suptitle(cellName)
    rmses[cellName] = rmsOpdIntData
    plt.pause(0.2)
plt.pause(0.5)
```

3.1.8 Generating new position lists to enable colocalized measurements on multiple systems.

```
# Copyright 2018-2020 Nick Anthony, Backman Biophotonics Lab, Northwestern University
#
# This file is part of PWSpy.
#
# PWSpy is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# PWSpy is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with PWSpy.  If not, see <https://www.gnu.org/licenses/>.
from pwspy.utility.micromanager import PropertyMap

from pwspy.utility.micromanager.positions import PositionList

"""This example demonstrates how to use generate new cell positions from a set of␣
↪positions after the sample has been picked up and likely shifted or rotated.
This method relies on measuring a set (at least 3) of reference positions before and␣
↪after moving the dish. You can then use these positions to generate an
affine transform. This affine transform can then be applied to your original cell␣
↪positions in order to generate a new set of positions for the same cells.
In the case of a standard cell culture dish it is best to use the corners of the glass␣
↪coverslip as your reference locations.

@author: Nick Anthony
"""

```

(continues on next page)

(continued from previous page)

```
import pathlib as pl
import matplotlib.pyplot as plt
from skimage.transform import AffineTransform
import numpy as np

plt.ion()
NCPath = pl.Path(r'Z:\Nick\NU-NC_EtOH fixation comparison\NC\Buccal cell')
NUPath = pl.Path(r'Z:\Nick\NU-NC_EtOH fixation comparison\NU\Buccal')

# Load the position list of the coverslip corners taken at the beginning of the␣
↪experiment.
preTreatRefPositions = PositionList.fromNanoMatFile(NCPath / 'corners_list2.mat',
↪'TIXYDrive')
# Load the position list of the coverslip corners after placing the dish back on the␣
↪microscope after treatment.
postTreatRefPositions = PositionList.fromPropertyMap(PropertyMap.loadFromFile(NUPath /
↪'corners.pos'))
# Generate an affine transform describing the difference between the two position lists.
transformMatrix = preTreatRefPositions.getAffineTransform(postTreatRefPositions)
# Load the positions of the cells we are measuring before the dish was removed.
preTreatCellPositions = PositionList.fromNanoMatFile(NCPath / 'cell_list.mat', 'TIXYDrive
↪')
# Transform the cell positions to the new expected locations.
postTreatCellPositions = preTreatCellPositions.applyAffineTransform(transformMatrix)
# Save the new positions to a file that can be loaded by Micro-Manager.
postTreatCellPositions.toPropertyMap().saveToFile(NUPath / 'transformedPositions.pos')

# Plot the reference and cell positions before treatment
fig, ax = plt.subplots()
preTreatRefPositions.plot(fig, ax)
preTreatCellPositions.plot(fig, ax)

# Plot the reference and cell positions after treatment
fig2, ax2 = plt.subplots()
postTreatRefPositions.plot(fig2, ax2)
postTreatCellPositions.plot(fig2, ax2)

af = AffineTransform(np.vstack([transformMatrix, [0, 0, 1]]))
print(f"Scale: {af.scale}, Shear: {af.shear}, Rotation: {af.rotation / (2*np.pi) * 360},␣
↪Translation: {af.translation}")
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z