CASCADe Documentation

Jeroen Bouwman

Created on : January, 2019

Last updated : February, 2019

More documents are freely available at xxx

Table of contents

List of figures

List of tables

At present several thousand transiting exoplanet systems have been discovered. For relatively few systems, however, a spectro-photometric characterization of the planetary atmospheres could be performed due to the tiny photometric signatures of the atmospheres and the large systematic noise introduced by the used instruments or the earth atmosphere. Several methods have been developed to deal with instrument and atmospheric noise. These methods include high precision calibration and modeling of the instruments, modeling of the noise using methods like principle component analysis or Gaussian processes and the simultaneous observations of many reference stars. Though significant progress has been made, most of these methods have drawbacks as they either have to make too many assumptions or do not fully utilize all information available in the data to negate the noise terms.

The CASCADe project implements a novel "data driven" method, pioneered by Schoelkopf et al (2015) utilizing the causal connections within a data set, and uses this to calibrate the spectral timeseries data of single transiting systems. The current code has been tested successfully to spectroscopic data obtained with the Spitzer and HST observatories.

Chapter 1

CASCADe documentation

1.1 Installing CASCADe

Clone a repository

To start working locally on an existing remote repository, clone it with the command *git clone <repository path>*. By cloning a repository, you'll download a copy of its files into your local computer, preserving the Git connection with the remote repository.

You can either clone it via HTTPS or [SSH](../ssh/README.md). If you chose to clone it via HTTPS, you'll have to enter your credentials every time you pull and push. With SSH, you enter your credentials once and can pull and push straightaway.

You can find both paths (HTTPS and SSH) by navigating to your project's landing page and clicking **Clone**. GitLab will prompt you with both paths, from which you can copy and paste in your command line.

As an example, consider a repository path:

- HTTPS: *https://gitlab.com/jbouwman/CASCADe*
- SSH: " git@gitlab.com:jbouwman/CASCADe "

To get started, open a terminal window in the directory you wish to clone the repository files into, and run one of the following commands.

Clone via HTTPS:

`bash git clone https://gitlab.com/jbouwman/CASCADe `

Clone via SSH:

`bash git clone git@gitlab.com:jbouwman/CASCADe `

Both commands will download a copy of the files in a folder named after the project's name.

You can then navigate to the directory and start working on it locally.

Go to the master branch to pull the latest changes from there

`bash git checkout master `

Download the latest changes in the project

This is for you to work on an up-to-date copy (it is important to do this every time you start working on a project), while you set up tracking branches. You pull from remote repositories to get all the changes made by users since the last time you cloned or pulled the project. Later, you can push your local commits to the remote repositories.

`bash git pull REMOTE NAME-OF-BRANCH `

When you first clone a repository, REMOTE is typically "origin". This is where the repository came from, and it indicates the SSH or HTTPS URL of the repository on the remote server. NAME-OF-BRANCH is usually "master", but it may be any existing branch.

1.2 Using Cascade

to run the code, first load all needed modules:

```
import cascade
```

Then, create transit spectroscopy object

```
tso = cascade.TSO.TSOSuite()
```

To reset all previous divined or initialized parameters

```
tso.execute("reset")
```

Initialize the TSO object using ini files which define the data, model parameters and behavior of the causal pixel model implemented in CASCADe.

```
path = cascade.initialize.default_initialization_path
tso = cascade.TSO.TSOSuite("initialize", "cascade_cpm.ini",
                           "cascade_object.ini",
                           "cascade_data_spectral_images.ini", path=path)
```

Load the observational data

```
tso.execute("load_data")
```

Subtract the background

```
tso.execute("subtract_background")
```

Sigma clip data

```
tso.execute("sigma_clip_data")
```

Determine the position of source from the spectroscopic data set

```
tso.execute("determine_source_position")
```

Set the extraction area within which the signal of the exoplanet will be determined

```
tso.execute("set_extraction_mask")
```

Extract the spectrum of the Star + planet in an optimal way

```
tso.execute("optimal_extraction")
```

Setup the matrix of regressors used to model the noise

```
tso.execute("select_regressors")
```

Define the eclipse model

```
tso.execute("define_eclipse_model")
```

Derive the calibrated time series and fit for the planetary signal

```
tso.execute("calibrate_timeseries")
```

Extract the planetary signal

```
tso.execute("extract_spectrum")
```

Correct the extracted planetary signal for non uniform subtraction of average eclipse/transit signal

```
tso.execute("correct_extracted_spectrum")
```

Save the planetary signal

```
tso.execute("save_results")
```

Plot results (planetary spectrum, residual etc.)

```
tso.execute("plot_results")
```

1.3 CASCADe API

1.3.1 The cascade.TSO module

The TSO module is the main module of the CASCADe package. The classes defined in this module define the time series object and all routines acting upon the TSO instance to extract the spectrum of the transiting exoplanet.

`class TSOSuite(`**init_files*, *path=None*`)`

Bases: `object`

Transit Spectroscopy Object Suite class.

This is the main class containing the light curve data of and transiting exoplanet and all functionality to calibrate and analyse the light curves and to extractthe spectrum of the transiting exoplanet.

Parameters `init_files` (*list* of *str*) – List containing all the initialization files needed to run the CASCADe code.

Raises `ValueError` – Raised when commands not recognized as valid

Examples

To make instance of TSOSuite class

```
>>> tso = cascade.TSO.TSOSuite()
```

`execute(`*command*, **init_files*, *path=None*`)`

Check if command is valid and excecute if True

Parameters

- `command` (*str*) – Command to be excecuted. If valid the method corresponding to the command will be excecuted
- `*init_files` (*tuple* of *str*) – Single or multiple file names of the .ini files containing the parameters defining the observation and calibration settings.
- `path` (*str*) – (optional) Filepath to the .ini files, standard value in None

Raises `ValueError` – error is raised if command is not valid

Examples

```
>>> tso.execute('reset')
```

`initialize_TSO(`**init_files*, *path=None*`)`

Initializes the TSO object by reading in a single or multiple .ini files

Parameters
- `*init_files` (*tuple* of *str*) – Single or multiple file names of the .ini files containing the parameters defining the observation and calibration settings.
- `path` (*str*) – (optional) Filepath to the .ini files, standard value in None

Variables `cascade_parameters` – cascade.initialize.initialize.configurator

Raises `FileNotFoundError` – Raises error if .ini file is not found

Examples

```
>>> tso.execute("initialize", init_flle_name)
```

`reset_TSO()`

Reset initialization of TSO object by removing all loaded parameters.

Examples

```
>>> tso.execute("reset")
```

`load_data()`

Load the transit time series observations from file, for the object, observatory, instrument and file location specified in the loaded initialization files

Variables `observation` (*cascade.instruments.instruments.Observation*) – Instance of Observation class containing all observational data

Examples

To load the observed data into the tso object:

```
>>> tso.execute("load_data")
```

`subtract_background()`

Subtract median background determined from data or background model from the science observations.

Variables `isBackgroundSubtracted` (*bool*) – *True* if background is subtracted

Raises `AttributeError` – In case no background data is defined

Examples

To subtract the background from the spectral images:

```
>>> tso.execute("subtract_background")
```

`static sigma_clip_data_cosmic(`*data*, *sigma*`)`

Sigma clip of time series data cube allong the time axis.

Parameters
- `data` (*ndarray*) – Input data to be cliped, last axis of data to be assumed the time
- `sigma` (*float*) – Sigma value of sigmaclip

Returns sigma_clip_mask (*ndarray*) – Updated mask for input data with bad data points flagged *(=1)*

`sigma_clip_data()`

Perform sigma clip on science data to flag bad data.

Variables `isSigmaCliped` (*bool*) – Set to *True* if bad data has been masked using sigma clip

Raises `AttributeError` – Error is raised if sigma value, the filter length or the convolution kernel are not defined.

Examples

To sigma clip the observation data stored in an instance of a TSO object, run the following example:

```
>>> tso.execute("sigma_clip_data")
```

`create_cleaned_dataset()`

Create a cleaned dataset to be used in regresion analysis.

Variables `cleaned_data` (*masked quantity*) – A cleaned version of the spctral timeseries data of the transiting exoplanet system

Raises *AttributeError, AssertionError* – An error is raised if the data set to be cleaned has not been background subtracted or no sigma clip has been run first to identify those pixels to be cleaned.

Examples

To create a cleaned version of the spectral data stored in an instance of a TSO object run:

```
>>> tso.execute("create_cleaned_dataset")
```

`define_eclipse_model()`

This function defines the light curve model used to analize the transit or eclipse. We define both the actual trasit/eclipse signal as wel as an calibration signal.

Variables

- `light_curve` (*ndarray*) – The lightcurve model
- `transit_timing` (*list*) – list with start time and end time of transit
- `light_curve_interpolated` (*list* of *ndarray*) – to the time grid of the observations interpolated lightcurve model
- `calibration_signal` (*list* of *ndarray*) – lightcurve model of the calibration signal
- `transittype` (*str*) – Currently either 'eclipse' or 'transit'

Raises `AttributeError` – Raises error if observations not properly defined.

Notes

The only lightcurve models presently incorporated in the CASCADe code are the ones provide by batman package.

Examples

To define the lightcurve model appropriate for the observations loaded into the instance of a TSO obkect, excecute the following command:

```
>>> tso.execute("define_eclipse_model")
```

`determine_source_position()`

This function determines the position of the source in the slit over time and the spectral trace.

We check if trace and position are already set, if not, determine them from the data by deriving the "center of light" of the dispersed light.

Variables

- `spectral_trace` (*ndarray*) – The trace of the dispersed light on the detector normalized to its median position. In case the data are extracted spectra, the trace is zero.
- `position` (*ndarray*) – Postion of the source on the detector in the cross dispersion directon as a function of time, normalized to the median position.
- `median_position` (*float*) – median source position.

Raises *AttributeError, AssertionError* – Raises error if input observational data or type of data is not properly difined. Also raises arror if data is not sigma cliped

---

Examples

To determine the position of the source in the cross dispersion direction from the in the tso object loaded data set, excecute the following command:

```
>>> tso.execute("determine_source_position")
```

`set_extraction_mask()`

Set mask which defines the area of interest within which a transit signal will be determined. The mask is set along the spectral trace with a pixel width of nExtractionWidth

Variables
- `nExtractionWidth` (*int*) – The width of the extraction aperture , cetered on the spectral trace of the source. In case of 1d spectral data, a width of 1 will be assumed.
- `extraction_mask` (*ndarray*) – In case data are Spectra : 1D mask In case data are Spectral images or cubes: 2D mask

Raises `AttributeError` – Raises error if the width of the mask or the source position and spectral trace are not defined.

Examples

To set the extraction mask, which will define the sub set of the data from which the planetary spectrum will be determined, sexcecute the following command:

```
>>> tso.execute("set_extraction_mask")
```

`_create_edge_mask`(*kernel*, *roi_mask_cube*)

Helper function for the optimal extraction task. This function creates an edge mask to mask all pixels for which the convolution kernel extends beyond the region of interest.

Parameters
- `kernel` (*array_like*) – Convolution kernel specific for a given instrument and observing mode, used in tasks such as replacing bad pixels and spectral extraction.
- `roi_mask` (*ndarray*) – Mask defining the region of interest from which the speectra will be extracted.

Returns **edge_mask** (*'array_like'*) – The edge mask based on the input kernel and roi_mask

`_create_extraction_profile`(*cleaned_data_with_roi_mask*, *extracted_spectra*, *kernel*, *mask_for_extraction*)

Helper function for the optimal extraction task. This function creates the normilzed source profile used for optimal extraction. The cleaned data is convolved with an appropriate kernel to smooth the profile and to increase the SNR. On the edges, where the kernel extends over the boundary, non convolved data is used to prevent edge effects.

Parameters
- `cleaned_data_with_roi_mask` (*numpy.ma.core.MaskedArray*) – The cleaned input data from which the extraction profile is derived. The mask attached to this data defines the region of interest around the target source.
- `extracted_spectra` (*numpy.ma.core.MaskedArray*) – Best guess for the spectrum of the source from which the extraction profile is determined.
- `kernel` (*ndarray*) – Convolution kernel used to create smoothed spectral images

- `mask_for_extraction` (*ndarray*) – Mask containing all pixels which are flagged as bad in the not cleaned data, i.e. all pixels which value have been replaced after cleaning.

Returns extraction_profile (*'ndarray'*) – The extraction profile used for optimal spectral extraction.

`_create_3dKernel(`*sigma_time*`)`

Helper function for the optimal extraction taks. This function creates a 3d kernel from a 2d instrument specific kernel to include the time dimention thus enabling convolution in both the spatial and wavelength direction as well as along the time axis.

Parameters `sigma_time` (*float*) –

Returns 3dKernel (*ndarray*)

Raises `AttributeError` – An error is raised if no instrument specific kernel is defined.

`optimal_extraction()`

Optimally extract spectrum using procedure of Horne 1986[1] The extraction consists of two iterations: The first one to determine the extraction profile, the second itereation to extract the spectrum of the target source.

Notes

We use a convolution with a kernel elongated along the spectral trace rather than a polynomial fit along the trace as in the original paper by Horne 1986.

Variables `dataset_optimal_extracted` (*cascade.data_model.SpectralDataTimeSeries*) – Time series if optimally extracted 1d spectra.

Raises *AttributeError, AssertionError* – An error is raised if the data and cleaned data sets are not defined, the source position is not determined or of the parameters for the optimal extraction task are not set in the initialization files.

References

Examples

To optimally extract a spectrum of the target star which data is stored in an TSO instance, excecute the following command:

```
>>> tso.execute("optimal_extraction")
```

`select_regressors()`

Select pixels which will be used as regressors.

Variables `regressor_list` (*list* of *int*) – List of regressors, using the following list index:

- first index: [# nod]
- second index: [# valid pixel in extraction mask]
- third index: [0=pixel coord; 1=list of regressors]
- forth index: [0=coordinate wave direction; 1=coordinate spatial direction]

Examples

To setup the list of regressors for each data point on which the exoplanet spectum will be based, execute the following command:

```
>>> tso.execute("select_regressors")
```

Horne 1986, PASP 98, 609

`static get_design_matrix(`*cleaned_data_in*, *original_mask_in*, *regressor_selection*, *nrebin*, *clip=False*, *clip_pctl_time=0.0*, *clip_pctl_regressors=0.0*`)`

Return the design matrix based on the data set itself

Parameters

- `cleaned_data_in` (*masked quantity*) – time series data with bad pixels corrected
- `original_mask_in` (*ndarray*) – data mask before cleaning
- `regressor_selection` (*list* of *int*) – List of index values of the data used as regressors
- `nrebin` (*int*) – Rebinning value for regressions [LEAVE at 1!!]
- `clip` (*bool*) – If 'True' bad regressors will be clipped out of selection
- `clip_pctl_time` (*float*) – Percentile of 'worst' regressors to be cut out in the time direction.
- `clip_pctl_regressors` (*float*) – Percentile of 'worst' regressors to be cut out in the wavelegth direction.

Returns *design_matrix* – The design matrix used in the causal pixel regression model

`reshape_data(`*data_in*`)`

Reshape the time series data to a uniform dimentional shape

Parameters `data_in` (*ndarray*) –

Returns data_out (*ndarray*)

`return_all_design_matrices(`*clip=False*, *clip_pctl_time=0.0*, *clip_pctl_regressors=0.0*`)`

Setup the regression matrix based on the sub set of the data slected to be used as calibrators.

Parameters

- `clip` (*bool*) – default False
- `clip_pctl_time` (*float*) – Default *0.00*
- `clip_pctl_regressors` (*float*) – Default *0.00*

Variables `design_matrix` (*list' of 'ndarray*) – list with design matrici with the following index convention:

- first index: [# nods]
- second index : [# of valid pixels within extraction mask]
- third index : [0]

Raises `AttributeError`

`calibrate_timeseries()`

This is the main function which runs the regression model to calibrate the input spectral light curve data and to extract the planetary signal as function of wavelength.

Variables `calibration_results` (*SimpleNamespace*) – The calibration_results attribute contains all calibrated data and auxilary data.

Raises `AttributeError` – an Error is raised if the nessecary steps to be able to run this task have not been executed properly or if the parameters for the regression model have not been set in the initialization files.

Examples

To create a calibrated spectral time series and derive the planetary signal execute the following command:

```
>>> tso.execute("calibrate_timeseries")
```

`extract_spectrum()`

Extract the planetary spectrum from the calibrated light curve data

Variables `exoplanet_spectrum` (*SimpleNamespace*) –

Raises `AttributeError`

Examples

To extract the exoplanet spectum from the calibrated signal, execute the following command:

```
>>> tso.execute("extract_spectrum")
```

`correct_extracted_spectrum()`
Make correction for non-uniform subtraction of transit signal due to differences in the relative weighting of the regressors

Raises AttributeError

Examples

To correct the extracted planetary signal for non-uniform subtraction of an averige transit depth, execute the following command:

```
>>> tso.execute("correct_extracted_spectrum")
```

`save_results()`
Save results

Raises AttributeError

Examples

To save the calibrated spectrum, execute the following command:

```
>>> tso.execute("save_results")
```

`plot_results()`
Plot the extracted planetary spectrum and scaled signal on the detector.

Raises AttributeError

Examples

To plot the planetary signal and other diadnostic plots, execute the following command:

```
>>> tso.execute("plot_results")
```

1.3.2 The cascade.cpm_model module

The cpm_model module defines the solver and other functionality for the regression model used in causal pixel model.

`solve_linear_equation(`*design_matrix, data, weights=None, cv_method='gcv', reg_par={'lam0': 1e-06, 'lam1': 100.0, 'nlam': 60}, feature_scaling='norm', degrees_of_freedom=None*`)`
Solve linear system using SVD with TIKHONOV regularization

Parameters

- `design_matrx` (*ndarray* with 'ndim=2') – Design matrix
- `data` (*ndarray*) – Data
- `weights` (*ndarray*) – Weights used in the linear least square minimization
- `cv_method` ((*'gvc'|'b95'|'B100'*)) – Method used to find optimal regularization parameter which can be:
  - 'gvc' : Generizalize Cross Validation [RECOMMENDED!!!],
  - 'b95' : normalized cumulatative periodogram using 95% limit,
  - 'B100': normalized cumulatative periodogram

- `reg_par` (*dict*) – Parameter describing search grid to find optimal regularization parameter lambda:
  - 'lam0' : minimum lambda
  - 'lam1' : maximum lambda,
  - 'nlam' : number of grid points
- `feature_scaling` ((*'norm'/None*)) – if the value is set to 'norm' all features are normalized using L2 norm else no featue scaling is applied.
- `degrees_of_freedom` (*int*) – Effective degrees_of_freedom, if set to None the value is calculated from the dimensions of the imput arrays.

Returns

fit_results (*tuple*) – In case the feature_scaling is set to None, the tuble contains the following parameters:

- (fit_parameters, err_fit_parameters, lam_reg)

else the following results are returned:

- (fit_parameters_scaled, err_fit_parameters_scaled, lam_reg, pc_matrix, fit_parameters, err_fit_parameters)

Notes

This routine solves the linear equation

$$Ax = y$$

by finding optimal solution ^x by minimizing

$$||y - A * \hat{x}||^2 + \lambda * ||\hat{x}||^2$$

For details on the implementation see[1],[2],[3],[4]

References

Examples

```
>>> import numpy as np
>>> from cascade.cpm_model import solve_linear_equation
>>> A = np.array([[1, 0, -1], [0, 1, 0], [1, 0, 1], [1, 1, 0], [-1, 1, 0]])
>>> coef = np.array([4, 2, 7])
>>> b = np.dot(A, coef)
>>> b = b + np.random.normal(0.0, 0.01, size=b.size)
>>> results = solve_linear_equation(A, b)
>>> print(results)
```

`return_PCR`(*design_matrix*, *n_components=None*, *variance_prior_scaling=1.0*)

Perform principal component regression with marginalization. To marginalize over the eigen-lightcurves we need to solve x = (A.T V^(-1) A)^(-1) * (A.T V^(-1) y), where V = C + B.T Lambda B, with B matrix containing the eigenlightcurves and lambda the median squared amplitudes of the eigenlightcurves.

1.3.3 The cascade.data_model module

This module defines the data models for the CASCADe transit spectroscopy code

PHD thesis by Diana Maria SIMA, "Regularization techniques in Model Fitting and Parameter estimation", KU Leuven 2006
Hogg et al 2010, "Data analysis recipies: Fitting a model to data"
Rust & O'Leaary, "Residual periodograms for choosing regularization parameters for ill-posed porblems"
Krakauer et al "Using generalized cross-validationto select parameters in inversions for regional carbon fluxes"

`class InstanceDescriptorMixin`

Bases: `object`

Mixin to be able to add descriptor to the instance of the class and not the class itself

`class UnitDesc(`*keyname*`)`

Bases: `object`

A descriptor for adding auxilary measurements, setting the property for the unit atribute

`class FlagDesc(`*keyname*`)`

Bases: `object`

A descriptor for adding logical flags

`class AuxilaryInfoDesc(`*keyname*`)`

Bases: `object`

A descriptor for adding Auxilary information to the dataset

`class MeasurementDesc(`*keyname*`)`

Bases: `object`

A descriptor for adding auxilary measurements, setting the properties for the the measurement and unit

`class SpectralData(`*wavelength=nan, wavelength_unit=None, data=nan, data_unit=None, uncertainty=nan, mask=False, **kwargs*`)`

Bases: *`cascade.data_model.data_model.InstanceDescriptorMixin`*

Class defining basic properties of spectral data In the instance if the SpectralData class all data are stored internally as numppy arrays. Outputted data are astropy Quantities unless no units (=None) are specified.

Parameters

- **`wavelength`** – wavelength of data (can be frequencies)
- **`wavelenth_unit`** – The physical unit of the wavelength (uses astropy.units)
- **`data`** – spectral data
- **`data_unit`** – the physical unit of the data (uses astropy.units)
- **`uncertainty`** – uncertainty on spectral data
- **`mask`** – mask defining masked data
- **`**kwargs`** – any auxilary data relevant to the spectral data (like position, detector temperature etc.) If unit is not explicitly given a unit atribute is added. Input argument can be instance of astropy quantity. Auxilary atributes are added to instance of the SpectralData class and not to the class itself. Only the required input stated above is always defined for all instances.

Examples

To create an instance of a SpectralData object with an initialization with data using units, run the following code:

```
>>> import numpy as np
>>> import astropu.units as u
>>> from cascade.data_model import SpectralData
```

```
>>> wave = np.array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0])*u.micron
>>> flux = np.array([8.0, 8.0, 8.0, 8.0, 8.0, 8.0])*u.Jy
>>> sd = SpectralData(wavelength=wave, data=flux)
```

```
>>> print(sd.data, sd.wavelength)
```

To change to convert the units to a different but equivalent unit:

```
>>> sd.data_unit = u.erg/u.s/u.cm**2/u.Hz
>>> sd.wavelength_unit = u.cm
>>> print(sd.data, sd.wavelength)
```

`wavelength`
The wavelength atttribute of the SpectralData is defined through a getter and setter method. This ensures that the returned wavelength has always a unit associated with it (if the wavelength_unit is set) and that the returned wavelength has the same dimension and mask as the data attribute.

`wavelength_unit`
The wavelength_unit attribute of the SpectralData is defined through a getter and setter method. This ensures that units can be updated and the wavelength value will be adjusted accordingly.

`data`
The data atttribute of the SpectralData is defined through a getter and setter method. In case data is initialized with a masked quantity, the data_unit and mask attributes will be set automatically.

`uncertainty`
The uncertainty atttribute of the SpectralData is defined through a getter and setter method. This ensures that the returned uncertainty has the same unit associated with it (if the data_unit is set) and the same mask as the data attribute.

`data_unit`
The data_unit attribute of the SpectralData is defined through a getter and setter method. This ensures that units can be updated and the data value will be adjusted accordingly.

`mask`
The mask atttribute of the SpectralData is defined through a getter and setter method. This ensures that the returned mask has the same dimension as the data attribute and will be set automatically if the input data is a masked array.

`class SpectralDataTimeSeries`(*wavelength=nan, wavelength_unit=None, data=array([[nan]]), data_unit=None, uncertainty=array([[nan]]), mask=False, time=nan, time_unit=None, **kwargs*)

Bases: *`cascade.data_model.data_model.SpectralData`*

Class defining timeseries of spectral data. This class inherits from SpectralData. The data stored within this class has one additional time dimension

Parameters

- `wavelength` (*'array_like'*) – wavelength assigned to each data point (can be also be frequencies)
- `wavelenth_unit` (*'astropy.units.core.Unit'*) – The physical unit of the wavelength .
- `data` (*'array_like'*) – The spectral data to be analysed. This can be either spectra (1D), spectral images (2D) or spectral data cubes (3D).
- `data_unit` (*astropy.units.core.Unit*) – The physical unit of the data.
- `uncertainty` – The uncertainty associated with the spectral data.
- `mask` (*'array_like'*) – The bad pixel mask flagging all data not to be used.
- `**kwargs` – any auxilary data relevant to the spectral data (like position, detector temperature etc.) If unit is not explicitly given a unit atribute is added. Input argument can be instance of astropy quantity. Auxilary atributes are added to instance of the SpectralData class and not to the class itself. Only the required input stated above is always defined for all instances.
- `time` (*'array_like'*) – The time of observation assiciated with each data point.
- `time_unit` (*'astropy.units.core.Unit'*) – physical unit of time data

---

Examples

To create an instance of a SpectralDataaTimeSeries object with an initialization with data using units, run the following code:

```
>>> import numpy as np
>>> from cascade.data_model import SpectralDataTimeSeries
```

```
>>> wave = np.array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0])*u.micron
>>> flux = np.array([8.0, 8.0, 8.0, 8.0, 8.0, 8.0])*u.Jy
>>> time = np.array([240000.0, 2400001.0, 2400002.0])*u.day
>>> flux_time_series = np.repeat(flux[:, np.newaxis], time.shape[0], 1)
>>> sdt = SpectralDataTimeSeries(wavelength=wave, data=flux_time_series,
                                 time=time)
```

`time`
The time atttribute of the SpectralDataTimeSeries is defined through a getter and setter method. This ensures that the returned time has always a unit associated with it if the time_unit is set and that the returned time has the same dimension and mask as the data attribute.

`time_unit`
The time_unit attribute of the SpectralDataTimeSeries is defined through a getter and setter method. This ensures that units can be updated and the time value will be adjusted accordingly.

1.3.4 The cascade.exoplanet_tools module

This Module defines the functionality to get catalog data on the targeted exoplanet and define the model ligth curve for the system. It also difines some usefull functionality for exoplanet atmosphere analysis.

`Kmag = Unit("Kmag")`
Definition of generic K band magnitude

`Vmag = Unit("Vmag")`
Definition of a generic Vband magnitude

`masked_array_input(`*func*`)`
Decorator function to check and handel masked Quantities

If one of the input arguments is wavelength or flux, the array can be a masked Quantity, masking out only 'bad' data. This decorator checks for masked arrays and upon finding the first masked array, passes the data and stores the mask to be used to create a masked Quantity after the function returns.

Parameters `func` (*method*) – Function to be decorated

`KmagToJy(`*magnitude: Unit("Kmag"), system='Johnson'*`)`
Convert Kband Magnitudes to Jy

Parameters
- `magnitude` (*'Kmag'*) – Input K band magnitude to be converted to Jy.
- `system` (*'str'*) – optional, either 'Johnson' or '2MASS', default is 'Johnson'

Returns **flux** (*'astropy.units.Quantity', u.Jy*) – Flux in Jy, converted from input Kband magnitude

Raises `AssertionError` – raises error if Photometric system not recognized

`JytoKmag(`*flux: Unit("Jy"), system='Johnson'*`)`
Convert flux in Jy to Kband Magnitudes

Parameters
- `flux` (*'astropy.units.Quantity', 'u.Jy or equivalent'*) – Input Flux to be converted K band magnitude.
- `system` (*'str'*) – optional, either 'Johnson' or '2MASS', default is 'Johnson'

Returns **magnitude** (*'astropy.units.Quantity', Kmag*) – Magnitude converted from input fkux value

Raises `AssertionError` – raises error if Photometric system not recognized

`Planck(`*wavelength: Unit("micron"), temperature: Unit("K")*`)`
This function calculates the emisison from a Black Body.

Parameters

- `wavelength` (*'astropy.units.Quantity'*) – Input wavelength in units of microns or equivalent
- `temperature` (*'astropy.units.Quantity'*) – Input temperature in units of Kelvin or equivalent

Returns blackbody (*'astropy.units.Quantity'*) – B_nu in cgs units [erg/s/cm2/Hz/sr]

Examples

```
>>> import cascade
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from astropy.visualization import quantity_support
>>> import astropy.units as u
```

```
>>> wave = np.arange(4, 15, 0.05) * u.micron
>>> temp = 300 * u.K
>>> flux = cascade.exoplanet_tools.Planck(wave, temp)
```

```
>>> with quantity_support():
...     plt.plot(wave, flux)
...     plt.show()
```

`SurfaceGravity`(*MassPlanet: Unit("jupiterMass"), RadiusPlanet: Unit("jupiterRad")*)

Calculates surface gravity of planet

Parameters

- `MassPlanet` – Mass of planet in units of Jupiter mass or equivalent
- `RadiusPlanet` – Radius of planet in units of Jupiter radius or equivalent

Returns *sgrav* – Surface gravity in units of m s-2

`ScaleHeight`(*MeanMolecularMass: Unit("u"), SurfaceGravity: Unit("m / s2"), Temperature: Unit("K")*)

Calculate the scaleheigth of the planet

Parameters

- `MeanMolecularMass` (*'astropy.units.Quantity'*) – in units of mass of the hydrogen atom or equivalent
- `SurfaceGravity` (*'astropy.units.Quantity'*) – in units of m s-2 or equivalent
- `Temperature` (*'astropy.units.Quantity'*) – in units of K or equivalent

Returns ScaleHeight (*'astropy.units.Quantity'*) – scaleheigth in unit of km

`TransitDepth`(*RadiusPlanet: Unit("jupiterRad"), RadiusStar: Unit("solRad")*)

Calculates the depth of the planetary transit assuming one can neglect the emision from the night side of the planet.

Parameters

- `Planet` (*Radius*) – Planetary radius in Jovian radii or equivalent
- `Star` (*Radius*) – Stellar radius in Solar radii or equivalent

Returns *depth* – Relative transit depth (unit less)

`EquilibriumTemperature`(*StellarTemperature: Unit("K"), StellarRadius: Unit("solRad"), SemiMajorAxis: Unit("AU"), Albedo=0.3, epsilon=0.7*)

Calculate the Equlibrium Temperature of the Planet

Parameters

- `StellarTemperature` (*'astropy.units.Quantity'*) – Temperature of the central star in units of K or equivalent
- `StellarRadius` (*'astropy.units.Quantity'*) – Radius of the central star in units of Solar Radii or equivalent
- `Albedo` (*'float'*) – Albedo of the planet.
- `SemiMajorAxis` (*'astropy.units.Quantity'*) – The semi-major axis of platetary orbit in units of AU or equivalent

- `epsilon` (*'float'*) – Green house effect parameter

Returns ET (*'astropy.units.Quantity'*) – Equlibrium Temperature of the exoplanet

`convert_spectrum_to_brighness_temperature(`*wavelength: Unit("micron"), contrast: Unit("%"), StellarTemperature: Unit("K"), StellarRadius: Unit("solRad"), RadiusPlanet: Unit("jupiterRad"), error: Unit("%") = None*`)`

Function to convert the secondary eclipse spectrum to brightness temperature.

Parameters
- `wavelength` – Wavelength in u.micron or equivalent unit.
- `contrast` – Contrast between planet and star in u.percent.
- `StellarTemperature` – Temperature if the star in u.K or equivalent unit.
- `StellarRadius` – Radius of the star in u.R_sun or equivalent unit.
- `RadiusPlanet` – Radius of the planet in u.R_jupiter or equivalent unit.
- `error` – (optional) Error on contrast in u.percent (standart value = None).

Returns
- *brighness_temperature* – Eclipse spectrum in units of brightness temperature.
- *error_brighness_temperature* – (optional) Error on the spectrum in units of brightness temperature.

`eclipse_to_transit(`*eclipse*`)`

Converts eclipse spectrum to transit spectrum

Parameters `eclipse` – Transit depth values to be converted

Returns *transit* – transit depth values derived from input eclipse values

`transit_to_eclipse(`*transit*`)`

Converts transit spectrum to eclipse spectrum

Parameters `transit` – Transit depth values to be converted

Returns *eclipse* – eclipse depth values derived from input transit values

`combine_spectra(`*identifier_list=[], path=''*`)`

Convienience function to combine multiple extracted spectra of the same source by calculating a weighted averige.

Parameters
- `identifier_list` (*'list' of 'str'*) – List of file identifiers of the individual spectra to be combined
- `path` (*'str'*) – path to the fits files

Returns combined_spectrum (*'array_like'*) – The combined spectrum based on the spectra specified in the input list

`get_calalog(`*catalog_name, update=True*`)`

Get exoplanet catalog data

Parameters
- `catalog_name` (*'str'*) – name of catalog to use
- `update` (*'bool'*) – Boolian indicating if local calalog file will be updated

Returns files_downloaded (*'list' of 'str'*) – list of downloaded catalog files

`parse_database(`*catalog_name, update=True*`)`

Read CSV files containing exoplanet catalog data

Parameters
- `catalog_name` (*'str'*) – name of catalog to use
- `update` (*'bool'*) – Boolian indicating if local calalog file will be updated

Returns table_list (*'list' of 'astropy.table.Table'*) – List containing astropy Tables with the parameters of the exoplanet systems in the database.

Note:

The following exoplanet databases can be used: The Transing exoplanet catalog (TEPCAT) The NASA exoplanet Archive The Exoplanet Orbit Database

Raises ValueError – Raises error if the input catalog is nor recognized

`extract_exoplanet_data`(*data_list*, *target_name_or_position*, *coord_unit=None*, *coordinate_frame='icrs'*, *search_radius=<Quantity 5. arcsec>*)

Extract the data record for a single target

Parameters

- `data_list` (*'list' of 'astropy.Table'*) – List containing table with exoplanet data
- `target_name_or_position` – Name of the target or coordinates of the target for which the record is extracted
- `coord_unit` – Unit of coordinates e.g (u.hourangle, u.deg)
- `coordinate_frame` – Frame of coordinate system e.g icrs

Returns table_list (*'list'*) – List containing data record of the specified planet

Examples

Download the Exoplanet Orbit Database:

```
>>> import cascade
>>> ct = cascade.exoplanet_tools.parse_database('EXOPLANETS.ORG',
                                                update=True)
```

Extract data record for single system:

```
>>> dr = cascade.exoplanet_tools.extract_exoplanet_data(ct, 'HD 189733 b')
>>> print(dr[0])
```

`class lightcuve`

Bases: object

Class defining lightcurve model used to model the observed transit/eclipse observations. Current valid lightcurve models: batman

Variables

- `lc` (*'array_like'*) – The lightcurve model
- `par` (*'ordered_dict'*) – The lightcurve parameters

Notes

Uses factory method to pick model/package used to calculate the lightcurve model.

Raises ValueError – Error is raised if no valid lightcurve model is defined

Examples

To test the generation of a ligthcurve model first generate standard .ini file and initialize cascade

```
>>> import cascade
>>> cascade.initialize.generate_default_initialization()
>>> path = cascade.initialize.default_initialization_path
>>> cascade_param = cascade.initialize.configurator(path+"cascade_default.ini")
```

Define the ligthcurve model specified in the .ini file

```
>>> lc_model = cascade.exoplanet_tools.lightcuve()
>>> print(lc_model.valid_models)
>>> print(lc_model.par)
```

Plot the normized lightcurve

```
>>> fig, axs = plt.subplots(1, 1, figsize=(12, 10))
>>> axs.plot(lc_model.lc[0], lc_model.lc[1])
>>> axs.set_ylabel(r'Normalized Signal')
>>> axs.set_xlabel(r'Phase')
>>> axes = plt.gca()
>>> axes.set_xlim([0, 1])
>>> axes.set_ylim([-1.1, 0.1])
>>> plt.show()
```

`valid_models = {'batman'}`

`class batman_model`

Bases: `object`

This class defines the lightcurve model used to analyse the observed transit/eclipse using the batman package by Laura Kreidberg[1].

Variables

- `lc` (*'array_like'*) – The values of the lightcurve model
- `par` (*'ordered_dict'*) – The model parameters difining the lightcurve model

---

References

`static define_batman_model(`*InputParameter*`)`

This function defines the light curve model used to analize the transit or eclipse. We use the batman package to calculate the light curves.We assume here a symmetric transit signal, that the secondary transit is at phase 0.5 and primary transit at 0.0.

Parameters `InputParameter` (*'dict'*) – Ordered dict containing all needed inut parameter to define model

Returns

- **tmodel** (*'array_like'*) – Orbital phase of planet used for lightcurve model
- **lcmode** (*'array_like'*) – Normalized values of the lightcurve model

`ReturnParFromIni()`

Get relevant parameters for lightcurve model from CASCADe intitialization files

Returns **par** (*'ordered_dict'*) – input model parameters for batman lightcurve model

`ReturnParFromDB()`

Get relevant parameters for lightcurve model from exoplanet database specified in CASCADe initialization file

Returns **par** (*'ordered_dict'*) – input model parameters for batman lightcurve model

Raises `ValueError` – Raises error in case the observation type is not recognized.

1.3.5 The cascade.initialize module

This Module defines the functionality to generate and read .ini files which are used to initialize CASCADe.

Examples

[1] Kreidberg, L. 2015, PASP 127, 1161

An example how the initilize module is used:

```
>>> import cascade
>>> default_path = cascade.initialize.default_initialization_path
>>> success = cascade.initialize.generate_default_initialization()
```

```
>>> tso = cascade.TSO.TSOSuite()
>>> print(cascade.initialize.cascade_configuration.isInitialized)
>>> print(tso.cascade_parameters.isInitialized)
>>> assert tso.cascade_parameters == cascade.initialize.cascade_configuration
```

```
>>> tso.execute('initialize', 'cascade_default.ini', path=default_path)
>>> print(cascade.initialize.cascade_configuration.isInitialized)
>>> print(tso.cascade_parameters.isInitialized)
```

```
>>> tso.execute("reset")
>>> print(cascade.initialize.cascade_configuration.isInitialized)
>>> print(tso.cascade_parameters.isInitialized)
```

`default_initialization_path = '/home/bouwman/CASCADeInit/'`
Default directory for CASCADe initialization files

`generate_default_initialization()`
Convenience function to generate an example .ini file for CASCADe initialization. The file will be saved in the default directory defined by default_initialization_path. Returns True if successfully runned.

`class configurator(`**file_names*`)`
Bases: `object`

This class defined the configuration singleton which will provide all parameters needed to run the CASCADe to all modules of the code.

`isInitialized = False`
Will be set to True if initialized

`reset()`
If called, this function will remove all initialized parameters.

`cascade_configuration = <cascade.initialize.initialize.configurator object>`
Singleton containing the entite configuration settings for the CASCADe code to work. This includes object and observation definitions and causal noise model settings.

1.3.6 The cascade.instruments module

Observatory and Instruments specific module of the CASCADe package

`class Observation`
Bases: `object`

This class handles the selection of the correct observatory and instrument classes and loads the time series data to be analyzed The observations specific parameters set during the initialization of the TSO object are used to select the observatory and instrument through a factory method and to load the specified observations into the instance of the TSO object.

Examples

The Observation calss is called during the following command:

```
>>> tso.execute("load_data")
```

`_Observation__check_observation_type()`
Function to check of the in the .ini specified observation type valid.

Raises `ValueError` – Raises error if the specified observation type is not valid or if the tso instance is not initialized.

`_Observation__do_observations(`*observatory*`)`
Factory method to load the needed observatory class and methods

`_Observation__get_observatory_name()`
Function to load the in the .ini files specified observatory name

Returns *ValueError* – Returns error if the observatory is not specified or recognized

`_Observation__valid_observation_type`
Set listing the current implemented observation types

`_Observation__valid_observatories`
Dictionary listing the current implemented observatories, used in factory method to select a observatory specific class

`class ObservatoryBase`
Bases: `object`

Observatory base class used to define the basic properties an observatory class should have

`name`
Name of the observatory.

`location`
Location of the observatory

`NAIF_ID`
NAIF ID of the observatory. With this the location relative to the sun and the observed target as a function of time can be determined. Needed to calculate BJD time.

`observatory_instruments`
The names of the instruments part of the observatory.

`class InstrumentBase`
Bases: `object`

Instrument base class used to define the basic properties an instrument class should have

`load_data()`
Method which allows to load data.

`get_instrument_setup()`
Method which gets the specific setup of the used instrument.

`name`
Name of the instrument.

`class HST`
Bases: *`cascade.instruments.instruments.ObservatoryBase`*

This observatory class defines the instuments and data handling for the spectropgraphs of the Spitzer Space telescope

`name`
Set to 'HST'

`location`
Set to 'SPACE'

`NAIF_ID`
Set to -48

`observatory_instruments`
Returns {'WFC3'}

`class HSTWFC3`

Bases: `cascade.instruments.instruments.InstrumentBase`

This instrument class defines the properties of the WFC3 instrument of the Hubble Space Telescope

For the instrument and observations the following valid options are available:

- detector subarrays : {'IRSUB128', 'IRSUB256', 'IRSUB512', 'GRISM128', 'GRISM256', 'GRISM512'}
- spectroscopic filters : {'G141'}
- imaging filters : {'F139M', 'F132N', 'F167N'}
- data type : {'SPECTRUM', 'SPECTRAL_IMAGE'}
- observing strategy : {'STARING'}
- data products : {'SPC', 'flt', 'COE'}

`name`

'WFC3'

Type Name of the HST instrument

`load_data()`

This function loads the WFC3 data form disk based on the parameters defined during the initialization of the TSO object.

`get_instrument_setup()`

Retrieve all relevant parameters defining the instrument and data setup

Returns par (*collections.OrderedDict*) – Dictionary containg all relevant parameters

Raises `ValueError` – If obseervationla parameters are not or incorrect defined an error will be raised

`get_spectra(`*is_background=False*`)`

This function combines all functionallity to read fits files containing the (uncalibrated) spectral timeseries, including orbital phase and wavelength information

Parameters `is_background` (*bool*) – if *True* the data represents an observaton of the IR background to be subtracted of the data of the transit spectroscopy target.

Returns SpectralTimeSeries (*cascade.data_model.SpectralDataTimeSeries*) – Instance of *SpectralDataTimeSeries* containing all spectroscopic data including uncertainties, time, wavelength and bad pixel mask.

Raises *AssertionError, KeyError* – Raises an error if no data is found or if certain expected fits keywords are not present in the data files.

`get_spectral_images(`*is_background=False*`)`

This function combines all functionallity to read fits files containing the (uncalibrated) spectral image timeseries, including orbital phase and wavelength information

Parameters `is_background` (*bool*) – if *True* the data represents an observaton of the IR background to be subtracted of the data of the transit spectroscopy target.

Returns SpectralTimeSeries (*cascade.data_model.SpectralDataTimeSeries*) – Instance of *SpectralDataTimeSeries* containing all spectroscopic data including uncertainties, time, wavelength and bad pixel mask.

Raises *AssertionError, KeyError* – Raises an error if no data is found or if certain expected fits keywords are not present in the data files.

`_define_convolution_kernel()`

Define the instrument specific convolution kernel which will be used in the correction procedure of bad pixels

`_define_region_of_interest()`

Defines region on detector which containes the intended target star.

`_get_background_cal_data()`

Get the calibration data from which the background in the science images can be determined.

Raises *FileNotFoundError, AttributeError* – An error is raised if the calibration images are not found or the background data is not properly defined.

Notes

For further details see:

http://www.stsci.edu/hst/wfc3/documents/ISRs/WFC3-2015-17.pdf

`_fit_background`(*science_data_in*)

Determes the background in the HST Grism data using a model for the background to the spectral timeseries data

Parameters `science_data_in` (*masked quantity*) – Input data for which the background will be determined

Returns SpectralTimeSeries (*SpectralDataTimeSeries*) – The fitted IR bacgound as a function of time

Notes

All details of the implemented model is described in:

http://www.stsci.edu/hst/wfc3/documents/ISRs/WFC3-2015-17.pdf

`_determine_relative_source_position`(*spectral_image_cube*, *mask*)

Determine the shift of the spectra (source) relative to the first integration. Note that it is important for this to work properly to have identified bad pixels and to correct the values using an edge preserving correction, i.e. an correction which takes into account the dispersion direction and psf size (relative to pixel size)

Parameters

- `spectral_image_cube` (*ndarray*) – Input spectral image data cube.
- `mask` (*ndarray* of *int*) – Bad pixel and region of interest mask. Values of 1 indicate flagged data.

Variables `relative_source_shift` – relative x and y position as a function of time.

`_determine_source_position_from_cal_image`(*calibration_image_cube*, *calibration_data_files*)

Determines the source position on the detector of the target source in the calibration image takes prior to the spectroscopic observations.

Parameters

- `calibration_image_cube` (*ndarray*) – Cube containing all acquisition images of the target.
- `calibration_data_files` (*list* of *str*) – List containing the file names associated with the calibraton data.

Variables `calibration_source_position` (*list' of 'tuple*) – The position of the source in the acquisition images associated with the HST spectral timeseries observations.

`_read_grism_configuration_files()`

Gets the relevant data from WFC3 configuration files

Variables

- `DYDX` (*list*) – The parameters for the spectral trace
- `DLDP` (*'list`*) – The parameters for the wavelength calibration

Raises `ValueError` – An error is raised if the parameters associated with the specified instrument mode can not be found in the calibration file.

`_read_reference_pixel_file()`

Read the calibration file containig the definition of the reference pixel appropriate for a given sub array and or filer

Variables `reference_pixels` (*collections.OrderedDict*) – Ordered dict containing the reference pixels to be used in the wavelength calibration.

`static _search_ref_pixel_cal_file(`*ptable*, *inst_aperture*, *inst_filter*`)`

Search the reference pixel calibration file for the reference pixel given the instrument aperture and filter.

Parameters
- `ptable` (*dict*) – Calibratrion table with reference positions
- `inst_aperture` (*str*) – The instrument aperture
- `inst_filter` (*str*) – The instrument filter

Returns
- **XREF** (*float*) – X reference position for the acquisition image
- **YREF** (*float*) – Y reference position for the acquisition image

Raises `ValueError` – An error is raises if the instrument aperture if filter is not fount in the calibration table

---

Notes

See also: http://www.stsci.edu/hst/observatory/apertures/wfc3.html

---

`_get_subarray_size(`*calibration_data*, *spectral_data*`)`

This function determines the size of the used subarray.

Parameters
- `calibration_data` –
- `spectral_data` –

Variables `subarray_sizes` –

Raises `AttributeError`

`_get_wavelength_calibration()`

The functions returns the wavelength calibration

Returns wave_cal (*ndarray*) – Wavelength calibraiton of the observations.

Raises `AttributeError` – An error is raised if the necessary calibration data is not yet defined.

`get_spectral_trace()`

Get spectral trace

Returns spectral_trace (*collections.OrderedDict*) – The spectral trace of the dispersed light (both position and wavelength)

Raises `AttributeError` – An error is raised in the necessary calibration data is not yet defined.

`static _WFC3Trace(`*xc*, *yc*, *DYDX*, *xref=522*, *yref=522*, *xref_grism=522*, *yref_grism=522*, *subarray=256*, *subarray_grism=256*`)`

This function defines the spectral trace for the wfc3 grism modes.

Parameters
- `xc` –
- `yc` –
- `DYDX` –
- `xref=522` –
- `yref=522` –
- `xref_grism=522` –
- `yref_grism=522` –
- `subarray=256` –
- `subarray_grism=256` –

Returns *trace*

Notes

Details can be found in:

http://www.stsci.edu/hst/wfc3/documents/ISRs/WFC3-2016-15.pdf

and

http://www.stsci.edu/hst/observatory/apertures/wfc3.html

`static _WFC3Dispersion`(*xc*, *yc*, *DYDX*, *DLDP*, *xref=522*, *yref=522*, *xref_grism=522*, *yref_grism=522*, *subarray=256*, *subarray_grism=256*)

Convert pixel coordinate to wavelength.

Parameters
- `xc` – X coordinate of direct image centroid
- `yc` – Y coordinate of direct image centroid
- `xref` –
- `yref` –
- `xref_grism` –
- `yref_grism` –
- `subarray` –
- `subarray_grism` –

Returns **wavelength** (*'astropy.units.core.Quantity'*) – return wavelength mapping of x coordinate in micron

Notes

For details of the method and coefficient adopted see[1] and[2]. See also:

http://www.stsci.edu/hst/wfc3/documents/ISRs/WFC3-2016-15.pdf

In case the direct image and spectral image are not taken with the same aperture, the centroid measurement is adjusted according to the table in:

http://www.stsci.edu/hst/observatory/apertures/wfc3.html

References

`class Spitzer`

Bases: *`cascade.instruments.instruments.ObservatoryBase`*

This observatory class defines the instuments and data handling for the spectropgraphs of the Spitzer Space telescope

`name`

Name of the observatory.

`location`

Location of the observatory

`NAIF_ID`

NAIF ID of the observatory. With this the location relative to the sun and the observed target as a function of time can be determined. Needed to calculate BJD time.

`observatory_instruments`

The names of the instruments part of the observatory.

`class SpitzerIRS`

Bases: *`cascade.instruments.instruments.InstrumentBase`*

This instrument class defines the properties of the IRS instrument of the Spitzer Space Telescope. For the instrument and observations the following valid options are available:

- detectors : {'SL', 'LL'}
- spectral orders : {'1', '2'}
- data products : {'droop', 'COE'}

Kuntschner et al. (2009)
Wilkins et al. (2014)

- observing mode : {'STARING', 'NODDED'}
- data type : {'SPECTRUM', 'SPECTRAL_IMAGE', 'SPECTRAL_DETECTOR_CUBE'}

`name`
Name of the instrument.

`load_data()`
Method which allows to load data.

`get_instrument_setup()`
Retrieve all relevant parameters defining the instrument and data setup

`get_spectra(`*is_background=False*`)`
This function combines all functionallity to read fits files containing the (uncalibrated) spectral time-series, including orbital phase and wavelength information

Parameters `is_background` (*bool*) – if *True* the data represents an observaton of the IR background to be subtracted of the data of the transit spectroscopy target.
Returns SpectralTimeSeries (*cascade.data_model.SpectralDataTimeSeries*) – Instance of *SpectralDataTimeSeries* containing all spectroscopic data including uncertainties, time, wavelength and bad pixel mask.
Raises *AssertionError, KeyError* – Raises an error if no data is found or if certain expected fits keywords are not present in the data files.

`get_spectral_images(`*is_background=False*`)`
This function combines all functionallity to read fits files containing the (uncalibrated) spectral time-series, including orbital phase and wavelength information

Parameters `is_background` (*bool*) – if *True* the data represents an observaton of the IR background to be subtracted of the data of the transit spectroscopy target.
Returns SpectralTimeSeries (*cascade.data_model.SpectralDataTimeSeries*) – Instance of *SpectralDataTimeSeries* containing all spectroscopic data including uncertainties, time, wavelength and bad pixel mask.
Raises *AssertionError, KeyError* – Raises an error if no data is found or if certain expected fits keywords are not present in the data files.

Notes

Notes on FOV: in the fits header the following relevant info is used:
- FOVID 26 IRS_Short-Lo_1st_Order_1st_Position
- FOVID 27 IRS_Short-Lo_1st_Order_2nd_Position
- FOVID 28 IRS_Short-Lo_1st_Order_Center_Position
- FOVID 29 IRS_Short-Lo_Module_Center
- FOVID 32 IRS_Short-Lo_2nd_Order_1st_Position
- FOVID 33 IRS_Short-Lo_2nd_Order_2nd_Position
- FOVID 34 IRS_Short-Lo_2nd_Order_Center_Position
- FOVID 40 IRS_Long-Lo_1st_Order_Center_Position
- FOVID 46 IRS_Long-Lo_2nd_Order_Center_Position

Notes on timing:

- FRAMTIME the total effective exposure time (ramp length) in seconds

`_define_convolution_kernel()`
Define the instrument specific convolution kernel which will be used in the correction procedure of bad pixels

`_define_region_of_interest()`
Defines region on detector which containes the intended target star.

`_get_order_mask()`
Gets the mask which defines the pixels used with a given spectral order

`_get_wavelength_calibration()`
Get wavelength calibration file

`get_detector_cubes(`*is_ background=False*`)`
This function combines all functionallity to read fits files containing the (uncalibrated) detector cubes (detector data on ramp level) timeseries, including orbital phase and wavelength information

Parameters `is_background` (*bool*) – if *True* the data represents an observaton of the IR background to be subtracted of the data of the transit spectroscopy target.

Returns SpectralTimeSeries (*cascade.data_ model.SpectralDataTimeSeries*) – Instance of *SpectralDataTimeSeries* containing all spectroscopic data including uncertainties, time, wavelength and bad pixel mask.

Raises *AssertionError, KeyError* – Raises an error if no data is found or if certain expected fits keywords are not present in the data files.

Notes

Notes on timing in header:

There are several integration-time-related keywords. Of greatest interest to the observer is the "effective integration time", which is the time on-chip between the first and last non-destructive reads for each pixel. It is called:

RAMPTIME = Total integration time for the current DCE.

The value of RAMPTIME gives the usable portion of the integration ramp, occurring between the beginning of the first read and the end of the last read. It excludes detector array pre-conditioning time. It may also be of interest to know the exposure time at other points along the ramp. The SUR sequence consists of the time taken at the beginning of a SUR sequence to condition the array (header keyword DEADTIME), the time taken to complete one read and one spin through the array (GRPTIME), and the non-destructive reads separated by uniform wait times. The wait consists of "clocking" through the array without reading or resetting. The time it takes to clock through the array once is given by the SAMPTIME keyword. So, for an N-read ramp:

RAMPTIME = 2x(N-1)xSAMPTIME

and

DCE duration = DEADTIME + GRPTIME + RAMPTIME

Note that peak-up data is not obtained in SUR mode. It is obtained in Double Correlated Sampling (DCS) mode. In that case, RAMPTIME gives the time interval between the 2nd sample and the preceeding reset.

`get_spectral_trace()`
Get spectral trace

1.3.7 The cascade.utilities module

This Module defines some utility functions used in cascade

`write_timeseries_to_fits(`*data*, *path*`)`
Write spectral timeseries data object to fits files

Parameters

- `data` (*'ndarry' or 'cascade.data_model.SpectralDataTimeSeries'*) – The data cube which will be save to fits file. For each time step a fits file will be generated.
- `path` (*'str'*) – Path to the directory where the fits files will be saved.

`find(`*pattern*, *path*`)`
Return a list of all data files

Parameters

- `pattern` (*'str'*) – Pattern used to search for files.
- `" 'str'` (*path*) – Path to directory to be searched.

Returns result (*'list' of 'str'*) – Sorted list of filenames matching the 'pattern' search

`spectres`(*new_spec_wavs*, *old_spec_wavs*, *spec_fluxes*, *spec_errs=None*)

SpectRes: A fast spectral resampling function. Copyright (C) 2017 A. C. Carnall Function for resampling spectra (and optionally associated uncertainties) onto a new wavelength basis.

Parameters

- `new_spec_wavs` (*numpy.ndarray*) – Array containing the new wavelength sampling desired for the spectrum or spectra.
- `old_spec_wavs` (*numpy.ndarray*) – 1D array containing the current wavelength sampling of the spectrum or spectra.
- `spec_fluxes` (*numpy.ndarray*) – Array containing spectral fluxes at the wavelengths specified in old_spec_wavs, last dimension must correspond to the shape of old_spec_wavs. Extra dimensions before this may be used to include multiple spectra.
- `spec_errs` (*numpy.ndarray (optional)*) – Array of the same shape as spec_fluxes containing uncertainties associated with each spectral flux value.

Returns

- **resampled_fluxes** (*numpy.ndarray*) – Array of resampled flux values, first dimension is the same length as new_spec_wavs, other dimensions are the same as spec_fluxes
- **resampled_errs** (*numpy.ndarray*) – Array of uncertainties associated with fluxes in resampled_fluxes. Only returned if spec_errs was specified.

Chapter 2

Indices and tables

- genindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W