MORTICIA Documentation

Release 1.0

D Griffith, A Ramkilowan

Mar 08, 2017

MORTICIA is Monte carlo Optical Rendering for Theatre Investigations of Capability under the Influence of the Atmosphere. It is intended to model optical spectrum imaging sensors observing targets through the atmosphere and to provide a statistical picture of the effectiveness of such sensors deployed in particular environments or geographical theatres.

Contents:

CHAPTER

ONE

INTRODUCTION

MORTICIA is Monte carlo Optical Rendering for Theatre Investigations of Capability under the Influence of the Atmosphere.

It is intended to model optical spectrum imaging sensors observing targets through the atmosphere and to provide a statistical picture of the effectiveness of such sensors deployed in particular environments or geographical theatres.

MORTICIA can be used to:

- Model a digital image recorded by a camera (imaging optical sensor) of a particular target under specific atmospheric conditions at a given time of day.
- Model imagery from ground-based, airborne or space optical sensors.
- Estimate the range of Detection, Recognition and Identification (DRI) for imaging optical sensors operating in the wavelength bands from the ultraviolet (~ 300 nm wavelength) to the thermal infrared (~ 14 microns wavelength).

CHAPTER

TWO

HOW TO USE MORTICIA

In order to use MORTICIA you will require a python installation (version 2.7 onwards). A Python 3.x distribution is still under development. A step-by-step guide will be provided on how to correctly use MORTICIA. From prerequisites, to installation instructions to running a simulation, providing links to nbMORTICIA examples etc

2.1 MORTICIA Dependencies

`MORTICIA` has been developed using the Anaconda distribution from Continuum Analytics and this is the recommended distribution for `MORTICIA` users. In principle, any Python 2.7 installation that can meet the dependencies should also work.

`MORTICIA` makes use of a variety of other Python packages. The most important of these are:

- numpy and scipy : The mainstay of technical computing in Python.
- matplotlib : Plotting.
- xray : A package for handling N-dimensional data arrays and datasets with named axes. Also for reading and writing data from netCDF format files.
- pandas : For handling time-series and data in a tabular/relational view.
- pint : for handling unit checking and conversion. Note that using `pint` for all numerical operations is not recommended and not done in `MORTICIA`. Rather, a `pint` Quantity object can be carried as metadata to an `xray.DataArray` object and checked (for example using Python `assert`) at object construction time or judiciously so that there is no significant performance impact of any unit checking.
- astropy and pyephem : for various astronomy-related calculations (sun, moon position).
- easygui is used for simple file/open and save dialogs.
- ipyparallel is used for parallel computation in IPython/Jupyter notebooks as well as other Python launch modes.
- dill will be required if running libRadtran on remote machines using `ipyparallel`
- paramiko will be required for authentication when using `ipyparallel` across machine boundaries

2.2 Installation and Requirements

`MORTICIA` has been developed largely in Python 2.7 and has not yet been tested in Python 3.X. A working installation of libRadtran is required to compute radiant environment maps and atmospheric transmittance. In the Monte carlo/statistical mode of operation, a compute cluster is generally required to achieve adequate sampling in a reasonable time. Parallel computation is performed using the ipyparallel package, which works in IPython/Jupyter notebooks as well as other Python launch modes. In principle, with `ipyparallel` it is possible to run the Python worker engines on any network-accessible compute resource. This means that it is possible to utilise a Linux-based compute cluster with a libRadtran installation in the background, while all foreground work is performed on a

Windows machine, for example. However, this setup may require significant setup to implement. All the required information to do this is provided in the `ipyparallel` documentation. Examples are provided in the *MORTICIA* notebooks in the nbMORITICIA repository on GitHub.

The most important top-level classes in `MORTICIA` are Sensor, Scene and Snap.

A Sensor comprises an imaging camera on a platform. A Scene comprises at least 2 scene elements (a target and a background). A Snap comprises a single integration period of the camera (snapshot), viewing the scene in a specific geometry (including range) under a specific set of atmospheric and illumination conditions. Illumination can be natural (sun, moon, stars) or artificial such as using a laser illuminator.

CHAPTER

THREE

THE MORTICIA STYLE GUIDE

The MORTICIA style guide is a comprehensive source of all things related to MORTICIA nomenclature. This guide is developed to be compatible with python 2.7, no immediate consideration is given to python 3.x at this stage. MORTICIA developers suspect that any changes for python 3.x will be subtle, unobtrusive and non-detrimental to MORTICIA development.

The style guide will be segmented into the following sections:

- Code formatting, in particular the use of PEP-8
- Code documenting, with a focus on docstring formatting
- Conventions indigenous to MORTICIA framework

3.1 To PEP-8 or not to PEP-8

While PEP-8, a high-level set of conventions has become the *de facto* standard for pythonic coding, it was deemed necessary by the MORTICIA developers to make certain exceptions. Similarly the PEP-257 is a set of guidelines for python docstrings.

A comprehensive PEP-8 guide can be found here, while a pdf PEP-8 cheat sheet can be found here.

The PEP-8 guides referenced above provides a thorough encapsulation of the do's and dont's of efficient and consistent python programming, it is however not all inclusive. There are topics which the guide does not address. This is in no way detrimental to any python development and attempting to fill these gaps are probably not an essential topic for the PEP-8 authors to address. Importantly, the PEP-8 is just a guide and one should not become indoctrinated with it's contents. To paraphrase Guido Van Rossum (Python's creator), it is important for ones python code to be consistent with PEP-8, it is more important to be consistent within a project, but it is most important to be consistent within one function or module.

The Python programming language is designed to be an *easy-to-read, easy-to-write* language. The PEP-8 style guide aims to accentuate this strong point of python by providing a means of consistency and optimal readability.

For sake of completeness the more commonly used cases of PEP-8 will be shown herein. It will be split into sub sections, after which a short section on MORTICIA's deviations will be noted. Where feasible a short discussion/ motivation together with an example will be provided

3.2 Code Layout

- ***Indentation*** [Use four space indentations. Most editors enforce this rule automatically. Spaces are preferred over tabs for indentation, tabs should be used however when adding to] code that already implements tabs for indentation (**consistency is key**).
- ***Blank lines*** [Two blank lines on either side of a top level function or class definition. Method definitions inside a class are surrounded by a single blank line. Use blank lines]

  sparingly inside function definitions to indicate logical sections.

```
        """docstrings enclosed by triple quotes go here.


        """

        import statements_go_here


        def top_level_func():
                does stuff
                return things


        class FooBar(object):
                """Class docstrings supplied for all public classes, functions and methods

                """


                def __init__(self, foo, bar):

                        initialise things in here
                        return

                def internal_func(self, bar, foo):
                        """note that method this local function is surrounded by a single blank l
                        """
                        perform some calculation/ data manipulation

                        return something


        # do stuff outside the class
```

- *Line length* : The PEP-8 style guide suggests a 79 column limit to all lines. The following is captured verbatim from the python PEP-8 style guide referenced earlier‘

  > “The limits are chosen to avoid wrapping in editors with the window width set to 80, even if the tool places a marker glyph in the final column when wrapping lines. Some web based tools may not offer dynamic line wrapping at all.
  >
  > Some teams strongly prefer a longer line length. For code maintained exclusively or primarily by a team that can reach agreement on this issue, it is okay to increase the nominal line length from 80 to 100 characters (effectively increasing the maximum length to 99 characters), provided that comments and docstrings are still wrapped at 72 characters....”
  >
  > The *MORTICIA* team have decided to conservatively follow the 99 character limit. Where possible and proves efficient line splitting is used.

3.3 Import Statements

This is a simple example:

```
import math
import seaborn as sns
```

3.4 Whitespace

3.5 Comments

3.6 Documentation Strings (docstrings)

3.7 Naming Conventions

This is an important one, the style of naming different python objects should make identifying the type of object intuitive

3.8 PEP-257

3.9 *From* Conventions Section

There is a data dictionary for `MORTICIA` which defines commonly used variable and data names.

xray.DataArray objects are created routinely, so the constructor is abbreviated to xD.

`MORTICIA` often deals with spectral variables, such as spectral transmission, spectral radiance etc. More often than not, these arrays are multidimensional. The *xray* package is used in *MORTICIA* for purposes of representing N-dimensional arrays (the *xray.DataArray* class).

However, *xray* does not (yet) support interpolation when the coordinates of the data hypercube are not the same and two or more *xray.DataArray* objects must be added, divided or multiplied.

Utility functions that operate on xray.DataArray objects are found in the *morticia.tools.xd* package. These functions perform axis harmonisation as well as unit checking and conversion.

The axis harmonisation functions for *xray.DataArray* objects requires that all axes have numeric coordinates. This applies also to axes that would benefit from having text labels. Text labels would be more appropriate for axes such as spectral channels and field orientation for MTF data. In the case of spectral channels (with axis label *chn*) will be channel numbers. These channel numbers would normally start from 0, but in some cases, such as when using a sub-range of correlated-k band models (e.g. *kato* or *fu* in libRadtran) the channel numbers would be a range of integers. The convention in `MORTICIA` is that the text labels then be placed in an axis attribute called *labels*.

This curently applies equally well to timeseries. For incorporation in multi-axis *xray.DataArray* objects, timestamps must be converted to Julian date. A list of labels for such axes can and should be maintained in the *xray.DataArray* attributes (*attrs*).

3.9.1 Units of Measure

Tracking of units of measure is not performed automatically in `MORTICIA` such as with the use of the pint package. However, many functions and classes in `MORTICIA` expect xray.DataArray instances that provide units of measure in the metadata. When units are provided in this way, they should be provided in manner consistent with the Python units package pint .

Commonly, when a `MORTICIA` function or method requires a scalar numeric input, it must be provided as a list with magnitude and units e.g. [30, 'mm']. Unitless quantities are provided as a simple numeric magnitude.

If a variable is named `x`, then the units for the variable can be stored as `units` in the xray.DataArray attributes dictionary (actually an OrderedDict). This units string should be recognizable to *pint* as a valid unit. When exporting xray.DataArray objects to NetCDF files in xray.Dataset objects, this convention of using the *units* attribute is recognised by NetCDF browsers and other NetCDF utilities. Further information on conventional NetCDF

attributes can be found at UCAR/Unidata. See the `MORTICIA` notebooks for examples of creating *xray* objects with unit attributes.

The UCAR/Unidata attributes for NetCDF file elements considered as highly recommended are

- *units*, units expressed as a string - SI units preferred
- *long_name*, such as may commonly be used to label graph axes
- *standard_name*, the standard name that may be used in a specific scientific community

and for `MORTICIA`, also

- *title*, such as may be used on the title of a plot of the data
- *labels*, such as may be used in the legends of a plot

The *standard_name* should come from the CF (Climate and Forecasting) name glossary, or the project should have its own vocabulary of short, standard and long names for all variables. These are to be found in the *morticia.moglo.py* module and may be expanded or updated from time to time.

A pint global unit registry is created when *morticia* or any sub-package is imported. Any other *morticia* paackages or modules share a single global unit registry called ureg. Convenience functions `Q_` for `Quantity` and `U_` for `Quantity(1.0, unit_str)` are also defined. Examples of usage are provided in the Jupyter notebooks/tutorials.

3.9.2 *From* Coordinate Systems

The general observer/target topocentric, cartesian coordinate system used in *MORTICIA* is +x towards North, +y towards the East and +z towards the zenith. Zenith angles are polar angles measured from the +z axis. This is a left-handed system and the alternative right-handed sytem is the same, except with +z towards nadir.

The global Earth Centered Earth Fixed (ECEF) coordinate system has the north pole in the +Z direction, the prime meridian (0 deg longitude) in the +X direction and the +Y direction at 90 deg east longitude. This is a right-handed coordinate system.

When dealing with sightlines and camera orientations, several conventions can be used. The Euler angles provide (relative to a given coordinate system, typically topocentric/geodetic) the view azimuth angle (VAZ = α) and view zenith angle (VZA = β). The camera roll is then the third Euler angle γ.

3.9.3 Logging Warnings and Exception Handling

As a rule, *MORTICIA* does not use logging to files. Preferably, if any checking is performed, exceptions are thrown. Informational messages should printed to the terminal using the *logging.info()* or *logging.debug()* calls. Warnings that the user should take action on are provided through *warnings.warn()*. If the warning may relate to the fact that *MORTICIA* code should be improved or debugged, it should be issued through *logging.warning()*. *MORTICIA* internal modules do not define any logging handlers or filters. This is left to the user.

3.10 General Terminology and Conventions

3.10.1 Camera and Imager

A *Camera* object in MORTICIA does *not* include a Lens - it is a camera body, including an FPA and a digitisation stage (which may be fully integrated into the FPA chip - a so-called “camera on a chip”). The *Imager* class incorporates a Camera and a Lens and is capable of producing actual imagery.

3.10.2 Modulation Transfer Function Calculation

For image modelling purposes, it is necessary to know the full 2D MTF, whereas the normal situation is that the MTF is described using horizontal and vertical MTF profile functions. Suppose that η and ξ are the spatial frequencies in the horizontal and vertical directions respectively and the $MTF_\eta(\eta)$ and $MTF_\xi(\xi)$ are the 1D MTFs in the horizontal and vertical directions. The 2D MTF is then computed as a rotationally weighted mean of the 1D MTFs as

$$MTF(\eta,\xi) = \frac{\eta^2 MTF_\eta\left(\sqrt{\eta^2+\xi^2}\right) + \xi^2 MTF_\xi\left(\sqrt{\eta^2+\xi^2}\right)}{\eta^2+\xi^2},$$

and if the sagittal spatial frequency is defined as $\rho = \sqrt{\eta^2+\xi^2}$, then

$$MTF(\eta,\xi) = \frac{\eta^2 MTF_\eta(\rho) + \xi^2 MTF_\xi(\rho)}{\rho^2}.$$

It is assumed here that the horizontal and vertical MTFs are symmetrical about the origin.

CHAPTER

FOUR

FOR THE DEVELOPERS

MORTICIA is an open-source project, we thus welcome contributions and or suggestions. If you are interested you may:

1. Fork us on Github
2. Subscribe to our mailing list here
3. Follow us on our twitter handle @MORTICIA

4.1 Structure

This section will disclose the structure of MORTICIA, and how different segments interface with each other. It is mainly aimed at developers not so much the users.

The most important top-level objects in MORTICIA are *Sensor* , *Scene* and Snap

CHAPTER

FIVE

PACKAGES

This is a sub-menu for all the packages

5.1 Sensor Package

The sensor package provides modelling of complete sensors, typically comprising an optical objective lens and a focal plane detector as a minimum (a camera). A sensor may also include a display and a human observer. The display and human observer are required in order to calculate ranges and probabilities of target detection, recognitions and identification (DRI).

Capability is also provided to model direct view systems comprising a telescope or binocular and human observer, without any electronics in the sensor chain.

5.1.1 Optics Module

The optics module provides modelling of basic objective lenses used by imaging systems. For the most part, these are optical systems that are near to diffraction-limited, with centred circular entrance pupils, possibly with a centred circular obscuration (as in a Cassegrain or Newtonian telescope).

class `optics.`**`Lens`**(*efl*, *fno*, *trn*, *wfe=None*, *obs=None*, *mtf=None*, *wvn_step=500.0*, *wfe_allowed=0.3*, *attrs=None*)

The Lens class encapsulates information and behaviour related to imaging lens systems. The chief characteristics of a lens are its spectral through-field, through-focus and through-frequency MTF, as well as the spectral transmission. In order to transform spatial frequencies in the image plane to angular spatial frequencies in object space, the effective focal length of the lens (efl) must also be known. The basic lens model is a near diffraction-limited system with a centred circular aperture having a centred circular obscuration (which may be absent), where the MTF is constant over the entire field of view (FOV). A lens with field-dependent MTF can be constructed by providing wavefront error input that varies with field.

The most basic lens model implemented here, from which more complicated lens models could inherit their properties have the following attributes

Parameters

- **`efl`** – The effective focal length of the lens in mm
- **`fno`** – The focal ratio of the lens
- **`trn`** – The spectral transmission of the lens (zero to unity).
- **`wfe`** – The wavefront error measured in waves. This can be a scalar, assumed the same for all wavelengths, or it can be provided as a function of wavelength and/or field position.
- **`obs`** – The obscuration ratio, being the ratio of the circular obscuration diameter to the full circular aperture aperture diameter
- **`mtf`** – The MTF of the lens. Either the MTF can be provided as a set of measurements or it can be computed from efl, fno, obs and wfe

The lens MTF is computed as a function of spatial frequency in the image, wavelength, defocus and field position.

The total RMS wavefront error is computed as

$$W = \sqrt{W_{\Delta z}^2 + W_a^2} = \sqrt{\left(\frac{\Delta z}{8\lambda F^2}\right)^2 + W_a^2}$$

where Δz is the defocus expressed in the same units as the wavelength λ, F is the focal ratio and W_a is the RMS waverfront error due to aberrations at best focus.

Lens constructor. The lens is constructed using the focal length, focal ratio and spectral transmittance, and optionally also the obscuration ratio and wavefront error.

Parameters

- **efl** – The effective focal length of the lens. The effective focal length must be a scalar value with units, e.g. [30, 'mm'].
- **fno** – The focal ratio (or f-number) which is th ratio of focal length to aperture diameter. This input must be a scalar value.
- **trn** (*xarray.DataArray*) – The spectral transmission function, typically a function of wavelength. The spectral transmittance function must be
- **wfe** – The wavefront error must be expressed in waves (unitless). It can be a function of wavelength (wvl) and field position (flr) , but not focus (fldz). If no input is provided, the wfe will default to zero over the same spectral region as the lens transmission function.
- **obs** – The obscuration ratio of the lens (scalar numeric) range 0.0 - 1.0, Default None (0.0)
- **mtf** – A complete pre-computed or measured MTF, with axes of spf, wvl, fldz, flr, flo
- **wvn_step** – Minimum spectral increment in wavenumbers (cm^-1) for MTF calculation. Default 500 cm^-1
- **wfe_allowed** – Maximum allowed wfe when computing lens defocus wfe, Default 0.5 waves at mean wavelength of transmission function domain.
- **attrs** – A dictionary of user-defined attributes and metadata. Consider including 'name' and 'long_name', 'title', 'summary' or other netCDF convention attributes.

Returns

__init__(*efl*, *fno*, *trn*, *wfe=None*, *obs=None*, *mtf=None*, *wvn_step=500.0*, *wfe_allowed=0.3*, *attrs=None*)

Lens constructor. The lens is constructed using the focal length, focal ratio and spectral transmittance, and optionally also the obscuration ratio and wavefront error.

Parameters

- **efl** – The effective focal length of the lens. The effective focal length must be a scalar value with units, e.g. [30, 'mm'].
- **fno** – The focal ratio (or f-number) which is th ratio of focal length to aperture diameter. This input must be a scalar value.
- **trn** (*xarray.DataArray*) – The spectral transmission function, typically a function of wavelength. The spectral transmittance function must be
- **wfe** – The wavefront error must be expressed in waves (unitless). It can be a function of wavelength (wvl) and field position (flr) , but not focus (fldz). If no input is provided, the wfe will default to zero over the same spectral region as the lens transmission function.
- **obs** – The obscuration ratio of the lens (scalar numeric) range 0.0 - 1.0, Default None (0.0)

- **mtf** – A complete pre-computed or measured MTF, with axes of spf, wvl, fldz, flr, flo
- **wvn_step** – Minimum spectral increment in wavenumbers (cm^-1) for MTF calculation. Default 500 cm^-1
- **wfe_allowed** – Maximum allowed wfe when computing lens defocus wfe, Default 0.5 waves at mean wavelength of transmission function domain.
- **attrs** – A dictionary of user-defined attributes and metadata. Consider including 'name' and 'long_name', 'title', 'summary' or other netCDF convention attributes.

Returns

mtf_obs_wfe()

Compute the multidimensional MTF of a Lens class object

See also:

•optics.atf which shows the method and formula by which the aberration/defocus MTF degradation is computed.

•optics.mtf_obs shows documents the typical calculation formulas for obscured MTF

optics.**atf**(*spf*, *wvl*, *fno*, *rms_wavefront_error*)

Compute the MTF degradation factor for a lens operating at the given wavelengths and and with the given focal ratios, RMS wavefront errors at the the specified spatial frequencies in the image plane. ATF stands for Aberration Transfer Function.

Parameters

- **spf** – Spatial frequencies in the image plane at which to compute the ATF. Spatial frequencies must be in reciprocal units to wavelengths i.e. if wavelengths are in mm, spatial frequencies must be in cycles per mm.
- **wvl** – Wavelengths at which to compute the ATF
- **fno** – Focal ratios at which to compute the ATF
- **rms_wavefront_error** – RMS wavefront error magnitudes (in waves) at which to compute the ATF

Returns A numpy array with the aberration transfer function.

Reference : Shannon, R.R., Handbook of Optics, Volume 1, 2nd Edition, Chapter 35 - Optical Specifications. "This is an approximation, however, and it becomes progressively less accurate as the amount of the rms wavefront error exceeds about 0.18 wavelength."

The formula used for computing the aberration MTF due to RMS wavefront error of W at spatial frequency f is

$$MTF_W(f) = 1 - \left(\frac{W}{0.18}\right)^2 \left[1 - 4\left(\frac{f}{f_c} - \frac{1}{2}\right)^2\right]$$

where the diffraction cutoff (or "critical") frequency is f_c.

See also:

optics.pmtf_obs_wfe

optics.**autocorr_circle**(*circle_radius*, *shift*)

Compute autocorrelation of a circular aperture of radius **circle_radius** with centre-to-centre sample displacements of **shift**.

Parameters

- **circle_radius** – Radius of circle
- **shift** – Centre-to-centre displacements at which to compute the autocorrelation

Returns Autocorrelation magnitude

See also:

optics.crosscorr_circle

optics.**crosscorr_circle**(*circle_radius*, *shift*)

Cross correlation of unit circle with circle of radius **circle_radius**. Compute the cross-correlation of a circular aperture of unit radius at the origin, with a circular aperture of radius **circle_radius**, with centre-to-centre displacements of **shift**.

Parameters

- **circle_radius** – Radius of circle to cross-correlate with unit circle (scalar numeric)
- **shift** – Centre-to-centre displacements at which to compute the cross-correlation

Returns Cross-correlation magnitude

See also:

optics.autocorr_circle

optics.**ctf_eye**(*spf*, *lum*, *w*, *num_eyes=2*, *formula=1*)

Compute the contrast transfer function of the human eye. By default, uses the condensed version of the Barten CTF.

Parameters

- **spf** – spatial frequencies in eye-space in cycles per milliradian (scalar or vector numpy array input)
- **lum** – mean luminance of the viewing area in cd/m^2 (scalar or vector numpy array input)
- **w** – the angular width of the viewing area, or the square root of the angular viewing area in square degrees (scalar or vector numpy input).
- **num_eyes** – The number of eyes used for viewing (2 for binocular viewing or 1 for monocular viewing). The default is num_eyes=2.
- **formula** – The formula variant used for the computation. Defaults (formula=1) to the simple formula first published by Barten in SPIE 2003. Other options are formula=11 and formula=14, which are slight variations.

Returns The CTF with respect to spf, lum and w (up to a 3D numpy array). Singular dimensions are squeezed out using numpy.squeeze().

optics.**mtf**(*spf*, *wvl*, *fno*)

Compute the simple (optimally focussed) diffraction Modulation Transfer Function (MTF) of a perfect lens with an unobscured circular aperture.

Parameters

- **spf** – Spatial frequencies in the image at which to compute the MTF
- **wvl** – Wavelength in units consistent with the spatial frequencies f
- **fno** – Focal ratio (working focal ratio) of the lens

Returns Modulation Transfer Function, with spatial frequency (spf) varying down columns and wavelength across rows If the frequencies are given in cycles per millimetre, the wavelengths must be in mm.

Any of the inputs can be a vector. The spatial frequencies are assigned to the rows of the output array, the wavelengths vary from column to column and Fno will vary in the third dimension, but singleton dimensions will be squeezed out.

See also:

optics.pmtf, optics.pmtf_obs, optics.pmtf_obs_wfe

optics.**mtf_obs**(*spf*, *wvl*, *fno*, *obs=0.0*)

Compute the optimally focussed diffraction Modulation Transfer Function of a perfect lens with an circular aperture having a centred circular obscuration. This is the monochromatic MTF computed at a number of discrete given wavelengths.

Parameters

- **spf** – Spatial frequencies in the image at which to compute the MTF (numpy vector).
- **wvl** – Wavelength in units consistent with the spatial frequencies **spf** (numpy vector)
- **fno** – Focal ratio (working focal ratio) of the lens (numpy vector).
- **obs** – The obscuration ratio (ratio of obscuration diameter to total aperture diameter). The obs input must be a scalar numeric.

Returns MTF with respect to spatial frequency, wavelength, focal ratio and obscuration ratio. Singleton dimensions are squeezed out.

optics.**n_air**(*wvl*, *temperature*, *pressure*)

Return the refractive index of air computed using the same formula used by ZEMAX See the section on Index of Refraction Computation in the Thermal Analysis chapter of the ZEMAX manual.

Parameters

- **wvl** – Wavelength(s) in microns. If all values of wvl exceed 100, then wavelengths are assumed to be in nm
- **temperature** – Temperature in Celsius.
- **pressure** – Relative air pressure (atmospheres, with 1 atm = 101 325 Pa).

Returns This function returns a matrix with wvl varying from row to row, temperature varying from column to column and pressure varying in the depth dimension. The returned matrix is subject to np.squeeze() to remove any singleton dimensions.

Reference : F. Kohlrausch, Praktische Physik, 1968, Vol 1, page 408

optics.**patf**(*spf*, *wvl*, *fno*, *rms_wavefront_error*, *wvl_weights*)

Compute the polychromatic aberration transfer function.

Parameters

- **spf** – Spatial frequencies in the image plane at which to compute the ATF. Spatial frequencies must be in reciprocal units to wavelengths i.e. if wavelengths are in mm, spatial frequencies must be in cycles per mm.
- **wvl** – Wavelengths at which to compute the ATF
- **fno** – Focal ratios at which to compute the ATF
- **rms_wavefront_error** – RMS wavefront error magnitudes (in waves) at which to compute the ATF
- **wvl_weights** – A numpy vector having the same length as the wvl vector, providing the relative weights of each of the wavelengths.

Returns Polychromatic Aberration Transfer Function in a numpy array.

See also:

optics.atf

optics.**pmtf**(*spf*, *wvl*, *fno*, *wvl_weights*)

Compute polychromatic MTF of lens at given spatial frequencies (in the image plane) for specified wavelengths and wavelength weighting factors.

Parameters

- **spf** – Spatial frequencies in the image at which to compute the polychromatic MTF (numpy vector).

- **wvl** – Wavelength in units consistent with the spatial frequencies **spf** (numpy vector)
- **fno** – Focal ratio (working focal ratio) of the lens (numpy vector).
- **wvl_weights** – A numpy vector having the same length as the wvl vector, providing the relative weights of each of the wavelengths.

Returns Polychromatic MTF with respect to spatial frequency, wavelength and focal ratio. Singleton dimensions are squeezed out of the returned numpy array.

optics.**pmtf_obs**(*spf*, *wvl*, *fno*, *wvl_weights*, *obs=0.0*)

Compute polychromatic MTF of obscured lens at given spatial frequencies (in the image plane) for specified wavelengths and wavelength weighting factors. The lens may have a circular obcuration of specific ratio.

Parameters

- **spf** – Spatial frequencies in the image at which to compute the polychromatic MTF (numpy vector).
- **wvl** – Wavelength in units consistent with the spatial frequencies **spf** (numpy vector)
- **fno** – Focal ratio (working focal ratio) of the lens (numpy vector).
- **wvl_weights** – A numpy vector having the same length as the wvl vector, providing the relative weights of each of the wavelengths.
- **obs** – Obscuration ratio

Returns Polychromatic obscured MTF with respect to spatial frequency, wavelength and focal ratio. Singleton dimensions are squeezed out of the returned numpy array.

optics.**pmtf_obs_wfe**(*spf*, *wvl*, *fno*, *rms_wavefront_error*, *wvl_weights*, *obs=0.0*)

Compute polychromatic modulation transfer function for lens having circular pupil with centred circular obscuration and with aberrations expressed in terms of RMS wavefront error.

Parameters

- **spf** – Spatial frequencies in the image at which to compute the polychromatic MTF (numpy vector).
- **wvl** – Wavelength in units consistent with the spatial frequencies **spf** (numpy vector)
- **fno** – Focal ratio (working focal ratio) of the lens (numpy vector).
- **rms_wavefront_error** – Numpy vector of RMS wavefront error values
- **wvl_weights** – A numpy vector having the same length as the wvl vector, providing the relative weights of each of the wavelengths.
- **obs** – Obscuration ratio (centred circular obscuration in circular pupil), ratio of obscuration diameter to aperture diameter.

Returns Numpy array with polychromatic modulation transfer function.

See also:

optics.atf, optics.pmtf_obs, optics.pmtf

5.1.2 Electronics Module

The electronics module provides modelling of all electronic components in the imaging chain. Inclusion of any electronic component into a sensor results in an "electro-optical" sensor. Electronic components include

- Solid state electro-optical detectors, including focal plane array (FPA) detectors
- Image intensifier tubes
- Image processing components
- Display systems

class electro.**Camera**(*fpa*, *ad_bit_depth*, *digital_gain=None*, *digital_offset=(0.0*, *'count')*, *noise=(0.0*, *'count')*, *sitf=None*, *exp_time_min=(1e-05*, *'s')*, *exp_time_max=(inf*, *'s')*, *attrs=None*)

The Camera class composes a FocalPlaneArray (FPA) together with a gain stage (signal transfer function), an analogue-to-digital converter of specific bit depth and an additional amplifier noise. The Camera converts photoelectrons to digital output data. NB : A Camera object in MORTICIA does *not* include the Lens. The most basic sensor object that includes a lens is an Imager object which includes a lens and a camera.

Camera constructor. This class if for representation of a Camera, which incorporates a FocalPlaneArray and a converter stage which converts photoelectrons into digital levels (also called digital numbers - DN and in MORTICIA, the pint unit 'count' is used).

Parameters

- **fpa** – The FocalPlaneArray incorporated into the Camera
- **ad_bit_depth** – Number of bits in the Analogue-to-Digital A/D converter. Must be provided in MORTICIA scalar format as e.g. [16, 'bit']
- **digital_gain** – The number of photoelectrons required to raise the output by 1 digital level (DN) Must be provided in MORTICIA scalar format as e.g. [2.2, 'e/count'].
- **digital_offset** – The digital level (DN) output of the camera for zero photoelectrons. This is not the "black level", which typically includes additional dark signal. Must be provided in MORTICIA scalar format as e.g. [10.0, 'count'].
- **noise** – An optional additional noise component to add to the signal. Must be provided as a MORTICIA scalar in either electrons or counts e.g. [2, 'e']. This is an RMS noise component, which is equivalent to a standard deviation.
- **sitf** – Signal transfer function (SiTF). As an alternative to providing the digital_gain and digital_offset, (particularly should the camera have essentially non-linear response) the SiTF can be provided as an xray.DataArray, with the input axis in units of electrons ('e') and the data in units of digital level ('count'). If digital_gain and digital_offset are provided, the SiTF is calculated and stored internally as an xray.DataArray.

:param :param attrs: A user-defined dictionary of other information about this Camera object. Could include items such as 'model', 'manufacturer' etc. A 'title' and 'long_name' are recommended attributes.

Returns

__init__(*fpa*, *ad_bit_depth*, *digital_gain=None*, *digital_offset=(0.0*, *'count')*, *noise=(0.0*, *'count')*, *sitf=None*, *exp_time_min=(1e-05*, *'s')*, *exp_time_max=(inf*, *'s')*, *attrs=None*)

Camera constructor. This class if for representation of a Camera, which incorporates a FocalPlaneArray and a converter stage which converts photoelectrons into digital levels (also called digital numbers - DN and in MORTICIA, the pint unit 'count' is used).

Parameters

- **fpa** – The FocalPlaneArray incorporated into the Camera
- **ad_bit_depth** – Number of bits in the Analogue-to-Digital A/D converter. Must be provided in MORTICIA scalar format as e.g. [16, 'bit']
- **digital_gain** – The number of photoelectrons required to raise the output by 1 digital level (DN) Must be provided in MORTICIA scalar format as e.g. [2.2, 'e/count'].
- **digital_offset** – The digital level (DN) output of the camera for zero photoelectrons. This is not the "black level", which typically includes additional dark signal. Must be provided in MORTICIA scalar format as e.g. [10.0, 'count'].
- **noise** – An optional additional noise component to add to the signal. Must be provided as a MORTICIA scalar in either electrons or counts e.g. [2, 'e']. This is an RMS noise component, which is equivalent to a standard deviation.

- **sitf** – Signal transfer function (SiTF). As an alternative to providing the digital_gain and digital_offset, (particularly should the camera have essentially non-linear response) the SiTF can be provided as an xray.DataArray, with the input axis in units of electrons ('e') and the data in units of digital level ('count'). If digital_gain and digital_offset are provided, the SiTF is calculated and stored internally as an xray.DataArray.

:param :param attrs: A user-defined dictionary of other information about this Camera object. Could include

items such as 'model', 'manufacturer' etc. A 'title' and 'long_name' are recommended attributes.

Returns

class electro.**FocalPlaneArray**(*pitch*, *aperture*, *pixels*, *wellcapacity*, *readnoise*, *darkcurrent*, *dsnu*, *prnu*, *sqe=None*, *asr=None*, *t_ref=(25.0, 'degC')*, *darkcurrent_delta_t=(7.0, 'delta_degC')*, *temperature=(25.0, 'degC')*, *attrs=None*)

Focal plane array detector. This implementation is typically at the chip level. That is, all or most of the information for building an FPA object can be found in the chip-level datasheet. The FPAs in question here are usually CCD, CMOS, scientific CMOS or electron-multiplying CCD (EMCCD).

The FPA class can be combined with an image intensifier tube (IIT) to produce an ICCD device.

This class does not model Time-Delay and Integration (TDI), which is a dynamic imaging process.

The class does allow for setting of FPA operating temperature and recalculation of dark current based on the dark current doubling delta temperature.

The FPA is a component of a Camera class object and has the basic function of converting photons to photoelectrons. FPA object can have multiple spectral channels as with a colour camera. The colour sampling spatial pattern can be as for a Bayer filter or a 3-CCD camera.

Constructor for FocalPlaneAArray objects

Parameters

- **pitch** – The centre-to-centre spacing of the FPA detector elements. This must be a list where the first element is the centre-to-centre spacing in the x-direction (or both x and y), the second element is the spacing in the y-direction (if different from the x-direction) and the last element is the units in which the pitch is provided (string). The units must be pint-recognizable.
- **aperture** – The effective pixel aperture of the FPA detector elements. This must also be a list, providing the x-aperture of the pixel, y-aperture (if different from x) and the units (string).
- **pixels** – A list of number of pixels in the x and y directions respectively
- **wellcapacity** – The well capacity of the pixel in number of electrons
- **readnoise** – The RMS read noise in electrons.
- **darkcurrent** – The dark current as a list, giving nagnitude and units. Units can be electrons per pixel per second (e/s), or in an amperage per pixel (A/s) or an amperage per unit area of the FPA (e.g. A/cm^2)
- **dsnu** – The dark signal non uniformity provided as a list giving the magnitude in the first element and units in the second element. The units can also be e/s, A or A/area. This is the standard deviation. The DSNU can also be specified as 0.0 or None.
- **prnu** – The photo-response non-uniformity. This is also a standard deviation and must be provided in units of '%'. That is, the prnu input is a list with the magnitude in the first position and the obligatory string '%' in the second position.

- **sqe** – This is a xray.DataArray object providing the Spectral Quantum Efficiency of the FPA. SQE must be provided with a single axis of wavelength values. The attribute sqe_units must be provided in the DataArray attributes and it must be dimensionless (the literal empty string ‘’). If any of the values exceeds unity, the SQE is assumed to be provided in percent. Alternatively, the asr input can be provided, but either the sqe or the asr input must be provided and not both, since they can be converted one from the other. SQE outside the spectral domain provided is assumed to be zero.
- **asr** – This is a xray.DataArray object providing the Absolute Spectral Response of the FPA. It can be provided as an alternative to the SQE. It must have the single axis of wavelength. Units are equivalent to A/W (photoelectron current per unit optical flux).
- **t_ref** – The reference temperature at which the dark current and other parameters are specified. To be provided as a [value, ‘units’] list. Default is [25.0, ‘degC’].
- **darkcurrent_delta_t** – The increase in temperature that causes doubling of the dark current. Default is [7.0, ‘delta_degC’].
- **temperature** – The operating temperature of the FocalPlaneArray. Default [25.0, ‘degC’]
- **attrs** – Dictionary of attributes and metadata. Entries with ‘name’, ‘long_name’, ‘title’, ‘summary’ or others, especially as per netCDF attribute conventions. The ‘manufacturer’ should possibly also be provided.

Returns

__init__(*pitch*, *aperture*, *pixels*, *wellcapacity*, *readnoise*, *darkcurrent*, *dsnu*, *prnu*, *sqe=None*, *asr=None*, *t_ref=(25.0*, *‘degC’)*, *darkcurrent_delta_t=(7.0*, *‘delta_degC’)*, *temperature=(25.0*, *‘degC’)*, *attrs=None*)

Constructor for FocalPlaneAArray objects

Parameters

- **pitch** – The centre-to-centre spacing of the FPA detector elements. This must be a list where the first element is the centre-to-centre spacing in the x-direction (or both x and y), the second element is the spacing in the y-direction (if different from the x-direction) and the last element is the units in which the pitch is provided (string). The units must be pint-recognizable.
- **aperture** – The effective pixel aperture of the FPA detector elements. This must also be a list, providing the x-aperture of the pixel, y-aperture (if different from x) and the units (string).
- **pixels** – A list of number of pixels in the x and y directions respectively
- **wellcapacity** – The well capacity of the pixel in number of electrons
- **readnoise** – The RMS read noise in electrons.
- **darkcurrent** – The dark current as a list, giving nagnitude and units. Units can be electrons per pixel per second (e/s), or in an amperage per pixel (A/s) or an amperage per unit area of the FPA (e.g. A/cm^2)
- **dsnu** – The dark signal non uniformity provided as a list giving the magnitude in the first element and units in the second element. The units can also be e/s, A or A/area. This is the standard deviation. The DSNU can also be specified as 0.0 or None.
- **prnu** – The photo-response non-uniformity. This is also a standard deviation and must be provided in units of ‘%’. That is, the prnu input is a list with the magnitude in the first position and the obligatory string ‘%’ in the second position.
- **sqe** – This is a xray.DataArray object providing the Spectral Quantum Efficiency of the FPA. SQE must be provided with a single axis of wavelength values. The attribute sqe_units must be provided in the DataArray attributes and it must be dimensionless (the literal empty string ‘’). If any of the values exceeds unity, the SQE is assumed to be provided in percent. Alternatively, the asr input can be provided, but either the sqe

or the asr input must be provided and not both, since they can be converted one from the other. SQE outside the spectral domain provided is assumed to be zero.

- **asr** – This is a xray.DataArray object providing the Absolute Spectral Response of the FPA. It can be provided as an alternative to the SQE. It must have the single axis of wavelength. Units are equivalent to A/W (photoelectron current per unit optical flux).
- **t_ref** – The reference temperature at which the dark current and other parameters are specified. To be provided as a [value, 'units'] list. Default is [25.0, 'degC'].
- **darkcurrent_delta_t** – The increase in temperature that causes doubling of the dark current. Default is [7.0, 'delta_degC'].
- **temperature** – The operating temperature of the FocalPlaneArray. Default [25.0, 'degC']
- **attrs** – Dictionary of attributes and metadata. Entries with 'name', 'long_name', 'title', 'summary' or others, especially as per netCDF attribute conventions. The 'manufacturer' should possibly also be provided.

Returns

compute_mtf()

Compute the MTF of a FocalPlaneArray. In this model, the FPA is assumed to be a rectangular array having pixels with rectangular aperture.

Returns

set_dark_current()

Set the dark current of the FocalPlaneArray (FPA) according to the dark current reference temperature and the current operating temperature of the FPA. :return:

class electro.**Imager**(*lens*, *camera*, *attrs=None*)

The Imager class encapsulates a complete imaging system with an optics.Lens and an electro.Camera. The Imager class can provide the digital outputs of the Lens+Camera when looking into a scene that provides the at-aperture (or at-sensor) radiance (the radiance apparent at the aperture of the Lens).

Constructor for Imager class.

Parameters

- **lens** – The optics.Lens for the Imager.
- **camera** – The electro.Camera for the Imager.
- **attrs** – User-defined dictionary of additional attributes, such as 'title', 'summary', 'long_name', 'manufacturer', especially those recommended for the netCDF conventions.

Returns an Imager object

__init__(*lens*, *camera*, *attrs=None*)

Constructor for Imager class.

Parameters

- **lens** – The optics.Lens for the Imager.
- **camera** – The electro.Camera for the Imager.
- **attrs** – User-defined dictionary of additional attributes, such as 'title', 'summary', 'long_name', 'manufacturer', especially those recommended for the netCDF conventions.

Returns an Imager object

electro.**xd_asr2sqe**(*asr*)

Convert absolute spectral response (ASR) to spectral quantum efficiency (SQE).

The conversion is performed through the Planck relationship

$$E = h\nu$$

where E is the photon energy, h is the planck constant and ν is the optical frequency. In terms of optical wavelength in a vacuum, the planck relation is

$$E = \frac{hc}{\lambda}$$

where c is the speed of light and λ is the wavelength.

Parameters **asr** – An xray.DataArray object providing the absolute spectral response (ASR). The DataArray must have a single axis providing the wavelength points, together with the standard attribute 'units' for the wavelength axis.

Returns Spectral Quantum Efficiency as an xray.DataArray object. Returned wavelengths will be 'nm' in the wavelength ('wvl') axis.

electro.**xd_sqe2asr**(*sqe*)

Convert spectral quantum efficiency (SQE) to absolute spectral response. This is the reverse conversion of that provided by electro.xd_asr2sqe.

Parameters **sqe** – The spectral quantum efficiency (SQE), provided as a xray.DataArray object in which SQE is provided as a function of wavelength. The wavelength axis must provide the 'units' attribute.

Returns Absolute spectral response (ASR) as an xray.DataArray object, havin a single axis of wavelength coordinates ('wvl'). Returned units for the wavelength axis will be 'nm'.

5.2 Radiometry Package

The radiometry package provides all MORTICIA functionality relating to radiometry and atmospheric radiative transfer. Atmospheric radiative transfer is performed by the libRadtran suite of tools.

5.2.1 LibRadTran Module

The LibRadTran module concerns itself with calculations for the radiant environment map. It solves the radiative transfer equation for a complex system.

- Item 1
- Item 2
- Item 3
- Item 4

class librad.**Case**(*casename=''*, *filename=None*, *optionlist=None*)

Class which encapsulates a run case of libRadtran/uvspec. This class has methods to read libRadtran/uvspec input files, write uvspec input files, run uvspec in parallel on multiple compute nodes and read uvspec output files. An important use-case is that of reading a uvspec input file called the "base case", altering the parameters of particular option keywords and then running the case and reading the outputs. This class is also used by the RadEnv class which encapsulates a radiant environment. Construction of radiant environment maps typically requires running an array of librad.Case instances.

Instantiate a libRadtran/uvspec case, typically by reading a uvspec .INP file.

Parameters

- **casename** – A user-defined name for the libRadtran/uvspec case
- **filename** – An optional filename from which to read the libRadtran/uvspec input

- **optionlist** – A list of option keywords and parameteres (tokens). The keyword existence is verified. Besides that, no error checking is performed automatically.

Returns None

```
>>> # Read a libRadtran/uvspec case from a .INP file and display the expanded input
>>> import morticia.rad.librad as librad
>>> libRadCase = librad.Case(filename='./examples/UVSPEC_AEROSOL.INP')  # Read uvspec input a
>>> print libRadCase   # This prints the uvspec input file, compare to contents of UVSPEC_AER
atmosphere_file ../data/atmmod/afglus.dat
source solar ../data/solar_flux/atlas_plus_modtran
mol_modify O3 300. DU
day_of_year 170
albedo 0.2
sza 32.0
rte_solver disort
number_of_streams 6
wavelength 299.0 341.0
slit_function_file ../examples/TRI_SLIT.DAT
spline 300 340 1
quiet
aerosol_vulcan 1
aerosol_haze 6
aerosol_season 1
aerosol_visibility 20.0
aerosol_angstrom 1.1 0.2
aerosol_modify ssa scale 0.85
aerosol_modify gg set 0.70
aerosol_file tau ../examples/AERO_TAU.DAT
```

__init__(*casename=''*, *filename=None*, *optionlist=None*)

Instantiate a libRadtran/uvspec case, typically by reading a uvspec .INP file.

Parameters

- **casename** – A user-defined name for the libRadtran/uvspec case
- **filename** – An optional filename from which to read the libRadtran/uvspec input
- **optionlist** – A list of option keywords and parameteres (tokens). The keyword existence is verified. Besides that, no error checking is performed automatically.

Returns None

```
>>> # Read a libRadtran/uvspec case from a .INP file and display the expanded input
>>> import morticia.rad.librad as librad
>>> libRadCase = librad.Case(filename='./examples/UVSPEC_AEROSOL.INP')  # Read uvspec inp
>>> print libRadCase   # This prints the uvspec input file, compare to contents of UVSPEC
atmosphere_file ../data/atmmod/afglus.dat
source solar ../data/solar_flux/atlas_plus_modtran
mol_modify O3 300. DU
day_of_year 170
albedo 0.2
sza 32.0
rte_solver disort
number_of_streams 6
wavelength 299.0 341.0
slit_function_file ../examples/TRI_SLIT.DAT
spline 300 340 1
quiet
aerosol_vulcan 1
aerosol_haze 6
aerosol_season 1
aerosol_visibility 20.0
aerosol_angstrom 1.1 0.2
aerosol_modify ssa scale 0.85
```

```
    aerosol_modify gg set 0.70
    aerosol_file tau ../examples/AERO_TAU.DAT
```

alter_option (*option*, *origin=('user', None)*)
Alter the parameters of a uvspec input option. If the option is not found, the option is appended with append_option instead.

Parameters

- **option** – List of keyword and tokens (parameters) to provide to the option keyword (list of strings).
- **origin** – A 2-tuple noting the "origin" of the change to this keyword. Default ('user', None)

Returns None

append_option (*option*, *origin=('user', None)*)
Append a libRadtran/uvspec options to this uvspec case. It will be appended at the end of the file

Parameters

- **option** – A list containing the keyword and keyword parameters (tokens). Boson keywords must be passed in "welded". e.g. x.append_option(['mol_modify O3', '270.0', 'DU'])
- **origin** – A 2-tuple giving the origin of the option and a "line number" reference. Default ('user', None) uvspec options.

Returns

del_option (*option*, *all=True*)
Delete a uvspec input option matching the given option.

Parameters

- **option** – Keyword of option to be deleted
- **all** – A flag indicating if all matching options must be deleted or only the first occurrence. The

default is to delete all matching occurrences. :return: True if an option was deleted or False if not

distribute_flux_data (*fluxdata*)
Distribute flux/user data read from uvspec output file to various data fields. This method will look at *output_user* options and attempt to assign flux/user data in a sensible way. .. note:

```
    There are potentially uvspec output formats that are not possible to process or to assign
    These are typically cases in which it is not possible to determine from the .INP and/or .
    how this data should be assigned.
```

Parameters **fluxdata** – Flux (irradiance) data read from uvspec output file

Returns None

irrad_units_str (*latex=False*)
Provide an irradiance units string e.g. W/m^2/nm. If output_quantity is set to 'brightness' or 'reflectivity', irrad_units will be 'K' or '' respectively.

Returns Irradiance units as a string

static merge_caselist_by_wavelength (*caselist*, *attr_name*)
Merges data in a particular attribute (property) of a list of librad.Case

This function can be used to merge data from a list of runs created using the function split_case_by_wavelength.

Parameters

- **caselist** – list of librad.Case after all cases have been run
- **attr_name** – name of attribute to merge e.g. 'edir'

Returns merged wavelengths, requested attribute as numnpy arrays

See also:

split_case_by_wavelength

prepare_for_keyword(*keyword*, *tokens*)
Make any possible preparations for occurrences of particular keywords :param keyword: The uvspec option keyword (string) :param tokens: The parameters (tokens) for the keyword as a list of strings :return:

prepare_for_mol_abs_param(*tokens*)
Make preparations for the desired molecular absorption parametrization. There are various molecular absorption options in libRadtran/uvspec and the options will certainly evolve. The default mode is to perform a spectral calculation (fine wavelength grid) with output of spectral radiances and irradiances. Some of the other modes, such as *crs* also output spectral data. The *crs* mode actually switches off molecular line absorption and considers only spectrally continuous scattering and absorption. This is really only good for the UV/blue spectrum. The *reptran_channel* mode and the correlated-k modes (*kato* variants, *fu*, *avhrr_kratz* and *lowtran*/*sbdart*) do not produce spectral radiances and irradiances. They produce band quantities which may even be summed using the *output_process sum* directive. A sub-range of correlated-k bins/channels can be selected using the *wavelength_index* directive.

See the libRadtran manual for further information on the relevant options. :return:

prepare_for_output_process(*tokens*)
Prepare for effects of the *output_process* keyword in the libRadtran/uvspec input file.

Parameters **tokens** – Keyword tokens of the uvspec *output_process* keyword.

Returns

prepare_for_polradtran()
Prepare for output from the polradtran solver

Returns

prepare_for_source(*tokens*)
Prepare for source, particularly units of various kinds, depending on the source. libRadtran/uvspec source options are mainly 'solar' and 'thermal' with some additional options for units.

Parameters **tokens** – uvspec 'source' keyword option parameters (tokens)

Returns None

process_outputs()
Process outputs from libRadtran into moglo.Scalar and xr.DataArray objects. Currently only radiance outputs are processed, along with a few typical flux outputs, such as *edir*.

Note that this method probably does not cover all libRadtran/uvspec inputs and outputs and will most likely continue to evolve, perhaps breaking existing code. :return: None

rad_units_str(*latex=False*)
Provide a radiance units string e.g. W/sr/m^2/nm. If output_quantity is set to 'brightness' or 'reflectivity', rad_units will be 'K' or '' respectively.

Returns Radiance units as a string

static read(*path*, *includes_seen=[]*)
Reads a libRadtran input file. This will construct the libRadtran case from the contents of the .INP file Adapted from code by libRadtran developers.

Parameters

- **path** – File path from which to read the uvspec input

- **includes_seen** – List of files already included (for recursion purposes to avoid infinite include loops)

Returns data, line_nos, path where data is the full data in the file with includes, line_nos shows the source of every line and path is the path to the main input file.

readerr (*filename=None*)

Read any error output from the uvspec run and attach it to the self object in the self.stderr property

It is important to check error output to see if the uvspec run was successful or what diagnostics were provided.

WARNING : If the 'verbose' option is used in the uvspec input file, the error output file can be VERY large. The 'verbose' option prints a large amount of information relating to setup of the RT problem. Please refer to the libRadtran manual in respect of using 'verbose'. It is encouraged in the beginning stages of setting up an RT problem to verify correctness, but with (perhaps) a single wavelength to reduce the stderr output volume.

Parameters **filename** – The name of te error file to read. If not provided, the default filename will be used. If provided as the empty string '', a file/open dialog will be presented.

Returns True if a file was read and False if the file was not found.

readout (*filename=None*)

Read uvspec output and assign to variables as intelligently as possible.

The general process of reading is:

1. If the user has specified output_user, just assume a flat file and read using np.loadtxt or np.genfromtxt.
2. If not user_output and the solver has no radiance blocks, assume a flat file and read using np.loadtxt. The variables to be read should be contained in the self.fluxline attribute.
3. Otherwise, if the output has radiance blocks, read those depending on the radiance block format for the specific solver. Keep reading flux and radiance blocks until the file is exhausted.

Once the data has all been read, the data is split up between the number of output levels and number of wavelengths. For radiance data, the order of numpy dimensions is *umu*, *phi*, *wavelength*, *zout* and *stokes*. That is, if a case has multiple zenith angles, multiple azimuth angles, multiple wavelengths and multiple output levels, the radiance property uu will have 4 dimensions. In the case of polradtran, there will be 5 dimensions to include the stokes parameters (if more than 1, which is the I = intensity parameter).

Output from uvspec depends on the solver and a number of other inputs, including the directive `output_user`. For the solvers ''disort, sdisort, spsdisort and presumably also disort2, the irradiance (flux) outputs default to

```
lambda edir edn eup uavgdir uavgdn uavgup
```

If radiances (intensities) have been requested with the umu (cosine zenith angles input), each line of flux data is followed by a block of radiance data as follows:

```
umu(0) u0u(umu(0))
umu(1) u0u(umu(1))
. . . .
. . . .
umu(n) u0u(umu(n))
```

where u0u is the azimuthally averaged radiance for the requested zenith angles.

If azimuth angles (phi) have also been specified, then the radiance block is extended as follows:

```
                              phi(0)          ...      phi(m)
      umu(0) u0u(umu(0)) uu(umu(0),phi(0)) ... uu(umu(0),phi(m))
      umu(1) u0u(umu(1)) uu(umu(1),phi(0)) ... uu(umu(1),phi(m))
      . . . .
      . . . .
      umu(n) u0u(umu(n)) uu(umu(n),phi(0)) ... uu(umu(n),phi(m))
```

Radiance outputs are not affected by output_user options.

For the polradtran solver, the flux block is as follows:

```
      lambda down_flux(1) up_flux(1) ... down_flux(iS) up_flux(iS)
```

where iS is the number of Stokes parameters specified using the 'polradtran nstokes' directive. If umu and phi are also specified, the radiance block is as follows:

```
                              phi(0)       ...      phi(m)
      Stokes vector I
      umu(0) u0u(umu(0)) uu(umu(0),phi(0)) ... uu(umu(0),phi(m))
      umu(1) u0u(umu(1)) uu(umu(1),phi(0)) ... uu(umu(1),phi(m))
      . . . .
      . . . .
      umu(n) u0u(umu(n)) uu(umu(n),phi(0)) ... uu(umu(n),phi(m))
      Stokes vector Q
      . . .
      . . .
```

The u0u (azimuthally averaged radiance) is always zero for polradtran.

For the two-stream solver (twostr), the flux block is

```
      lambda edir edn eup uavg
```

The directive keyword `brightness` can also change output. The documentation simply states that radiances and irradiances are just converted to brightness temperatures.

The keyword directive `zout` and it's parameters will influence output format as well. In general the output is repeated for each given value of `zout` or `zout_sea`.

The keyword directive `output` and its parameters will also have a major effect.

The `output sum` keyword sums output data over the wavelength dimension. This in contrast to `output integrate`, which performs a spectral integral.

The keyword directive `header` should not be used at all. This produces some header information in the output that will cause errors. A warning is issued of the `header` keyword is used in the input.

Parameters **`filename`** – File from which to read the output. Defaults to name of input file, but with the .OUT

extension. :return: None

`run`(*stderr_to_file=True*, *write_input=True*, *read_output=True*, *block=True*, *purge=True*, *check_output=False*)
Run the libRadtran/uvspec case.

This will run the libRadtran/uvspec Case instance provided. Some control is provided regarding the handling of the standard error output from uvspec. This function only returns when uvspec terminates.

Parameters

- **`stderr_to_file`** – Controls whether standard error output goes to the screen or is written to a file having the same name as the input/output files, except with the extension .ERR. Default is true. Any error output
- **`write_input`** – Controls whether the input file is written out before execution. Default is True.

- **read_output** – Controls whether the output file is read after execution. Default is True.
- **block** – By default, this method waits until uvspec terminates. If set False, the uvspec process is released to background and read_output is set to False (regardless of user input).
- **purge** – If set True, the input and output files from this run will be deleted after the run is complete and the outputs have been read. Will only be honoured if read_output is also True. Default is True. To prevent purging for a particular librad.Case, set the individual case property to False.
- **check_output** – If set True, the uvspec command is executed using the subprocess.check_output call, which will place the standard output from the run in self.check_output. This is useful for diagnostic purposes.

Returns Returns self. This is important for running across networks.

set_option(**superlist*)

Set a uvspec input option with flexible input format

Parameters **superlist** – All set_option arguments are collected into a list which are coverted to strings (if not already strings), splitting each resulting string at spaces.

Returns None

Example:

```
>>> import morticia.rad.librad as librad
>>> libRadCase = librad.Case(casename='MyExample')  # Creates a blank libRadtran/uvspec c
>>> libRadCase.set_option('source solar', '../data/solar_flux/atlas_plus_modtran')
>>> libRadCase.set_option('wavelength', 300, 400)  # can provide literal numerics
>>> print libRadCase
```

set_view_geometry(*sza=0.0*, *saa=0.0*, *oza=0.0*, *oaa=0.0*)

Set the Case view geometry relative to the sun

Sets the sun and observer zenith and azimuth angles. Note that libRadtran/uvspec sets the light propagation azimuth angle in the `phi` and `phi0` inputs. This is 180^circ different in azimuth to the direction of viewing. The `phi0` input is the direction of solar light propagation from the sun and is 180° from the saa input here. However, the `phi` input is the direction of light propagation from target to sensor and is therefore *the same* as the oaa input to this function.

The `umu` input to libRadtran/uvspec is the cosine of the zenith angle of light propagation from target to sensor. Hence `umu` is positive for downward-looking (upward-propagating) cases.

Also, from the libRadtran manual : phi = phi0 indicates that the sensor looks into the direction of the sun, while phi-phi0 = 180 deg means that the sun is behind the sensor.

Parameters

- **sza** – solar zenith angle in degrees from the zenith (range 0 to 90 degrees)
- **saa** – solar azimuth angle in degrees from north through east (range zero to)
- **oza** – observation zenith angle in degrees from the zenith. Note that this is the zenith angle of a vector pointing from the target/scene to the observer (satellite/sensor).
- **oaa** – observation azimuth angle in degrees from north through east. Note again that this is the azimuth angle of a vector pointing from the target/scene to the observer.

Returns None

set_wavelength_grid(*wvl_grid*)

Set the internal RT solver wavelength grid for this librad.Case

This method sets the internal grid for the RT calculations in libRadtran. Study the libRadtran manual before trying to use this option. Once this function has been used, the option wavelength_grid_file will

be activated and the grid file will be *casename_wvl_grid.dat*, where the casename is the name of the librad.Case. When the case is Run, the wavelength grid file will be written out to the current directory before uvspec is executed. By default (unless purge is disabled), the file will also automatically be deleted once the tun terminates.

Parameters **wvl_grid** – An np.array of floats that provide the wavelengths in nm at which to perform the internal radiative transfer calculation.

Returns None

split_case_by_wavelength(*n_sub_ranges*, *overlap*)

Split a librad.Case into a list of cases with wavelength sub-ranges This function can be useful to distribute a case over a compute cluster. Equal wavelength interval sub-cases should have similar runtimes. Use ipyparallel to run the list on a cluster. If the wavelength range is sufficiently large, consider setting n_sub_ranges to the number of available processors.

This function only works with a librad.Case that uses the *wavelength* option keyword. Cases that make use of other methods to set the wavelength range will not work.

Parameters

- **n_sub_ranges** – Number of cases in the list, which is also the number of wavelength sub-ranges.
- **overlap** – Amount of wavelength overlap between subrange cases.

Returns List of librad.Case

See also:

merge_caselist_by_wavelength

unset_wavelength_grid()

Remove the wavelength grid file and associated grid data for this librad.Case.

Returns None

write(*filename=''*)

Write libRadtran/uvspec input to a file (.INP extension by default. If the filename input is given as '', a file save dialog will be presented

Parameters **filename** – Filename to which to write the uvspec input

Returns

class librad.**HyperRadEnv**(*base_case*, *n_pol*, *n_azi*, *hyper_axes*, *mxumu=48*, *mxphi=19*, *hemi=False*, *n_sza=0*)

A higher dimensional form of radiant environment map REM. This class inherits from RadEnv. A RadEnv instance has mandatory radiance data dimensions of propagation zenith angle (pza), propaation azimuth angle (paz), wavelength, height above surface (zout) and stokes parameter. These are the outputs that can all be obtained from a single run of libRadtran. The HyperRadEnv allows for increasing the dimensionality of the REM dataset using other (typically scalar) inputs to libRadtran. Most significant amongst these are solar zenith angle (sza), surface albedo (albedo) and surface temperature (sur_temperature). Solar zenith angle and albedo are only applicable in *source solar* runs, while sur_temperature is only applicable in *source thermal* REMs.

Other possible higher dimensions that can be added include: Aerosol loading as specified by visibility or optical depth

Create a set of uvspec runs covering the whole sphere to calculate a full radiant environment map. Where the base_case is the uvspec case on which to base the environmental map, Name is the name to give the environmental map and n_pol and n_azi are the number of polar and azimuthal sightline angles to generate. The mxumu and mxphi are the maximum number of polar and azimuth angles to calculate in a single run of uvspec. The default values are mxumu = 48, and mxphi = 19. These values are taken from the standard libRadtran distribution (/libsrc_f/DISORT.MXD) maximum parameter file. If using the polradtran solver, the corresponding file is /libsrc_f/POLRADTRAN.MXD. Other solvers may have different restrictions. A warning will be issued if the solver is not in the DISORT/POLRADTRAN family.

Parameters

- **base_case** – librad.Case object providing the case on which the environment map is to be based. Note that not any basecase can be used. As a general guideline, the basecase should have standard irradiance outputs (i.e. should not use the *output_user* keyword). It should also not the use *output_process* or *output_quantity* keywords, which change the units and/or format of the libRadtran/uvspec output.

  Minimal validation of the basecase is performed. However, use with *mol_abs_param* such as *kato* and *fu* is important for *MORTICIA* and these are supported (k-distribution or *correlated-k* parametrizations). Use of *output_process per_nm* is appropriate for *source thermal* REMs to get radiance units per nanometre rather than per inverse cm.

- **n_pol** – Number of polar angles (view/propagation zenith angles)
- **n_azi** – Number of azimuthal angles.
- **hyper_axes** – Dictionary of libRadtran keywords, with a list (or numpy array) of values to assign to that keyword to create multiple copies of the REM e.g. {'sza': [0.0 30.0 45.0]} will create 3 runs of the REM with these solar zenith angles. Note that the number of REMs goes up very rapidly. If 3 surface albedo values as well as 3 solar zenith angles, the compute overhead goes up by a factor of 9. In general, the keywords should have a scalar and numeric input (such as *albedo*, *sza* and *sur_temperature*). This input parameter is mandatory and there is no default.
- **mxumu** – Maximum number of polar angles per case.
- **mxphi** – Maximum number of azimuthal angles per case.
- **hemi** – If set True, will generate only a single hemisphere being on one side of the solar principle plane. Default is False i.e. the environment map covers the full sphere. Note that if hemi=True, the number of REM samples in azimuth becomes n_azi $\times$ 2. This is the recommended mode (hemi=True) for MORTICIA purposes, since it reduces execution time.
- **n_sza** – The number of solar zenith angles (SZA) at which to perform transmittance and path radiance computations. Each SZA will result in another run of the base case (no radiances)

__init__(*base_case*, *n_pol*, *n_azi*, *hyper_axes*, *mxumu=48*, *mxphi=19*, *hemi=False*, *n_sza=0*)

Create a set of uvspec runs covering the whole sphere to calculate a full radiant environment map. Where the base_case is the uvspec case on which to base the environmental map, Name is the name to give the environmental map and n_pol and n_azi are the number of polar and azimuthal sightline angles to generate. The mxumu and mxphi are the maximum number of polar and azimuth angles to calculate in a single run of uvspec. The default values are mxumu = 48, and mxphi = 19. These values are taken from the standard libRadtran distribution (/libsrc_f/DISORT.MXD) maximum parameter file. If using the polradtran solver, the corresponding file is /libsrc_f/POLRADTRAN.MXD. Other solvers may have different restrictions. A warning will be issued if the solver is not in the DISORT/POLRADTRAN family.

Parameters

- **base_case** – librad.Case object providing the case on which the environment map is to be based. Note that not any basecase can be used. As a general guideline, the basecase should have standard irradiance outputs (i.e. should not use the *output_user* keyword). It should also not the use *output_process* or *output_quantity* keywords, which change the units and/or format of the libRadtran/uvspec output.

  Minimal validation of the basecase is performed. However, use with *mol_abs_param* such as *kato* and *fu* is important for *MORTICIA* and these are supported (k-distribution or *correlated-k* parametrizations). Use of *output_process per_nm* is appropriate for *source thermal* REMs to get radiance units per nanometre rather than per inverse cm.

- **n_pol** – Number of polar angles (view/propagation zenith angles)
- **n_azi** – Number of azimuthal angles.
- **hyper_axes** – Dictionary of libRadtran keywords, with a list (or numpy array) of values to assign to that keyword to create multiple copies of the REM e.g. {'sza': [0.0 30.0 45.0]} will create 3 runs of the REM with these solar zenith angles. Note that the number of REMs goes up very rapidly. If 3 surface albedo values as well as 3 solar zenith angles, the compute overhead goes up by a factor of 9. In general, the keywords should have a scalar and numeric input (such as *albedo*, *sza* and *sur_temperature*). This input parameter is mandatory and there is no default.
- **mxumu** – Maximum number of polar angles per case.
- **mxphi** – Maximum number of azimuthal angles per case.
- **hemi** – If set True, will generate only a single hemisphere being on one side of the solar principle plane. Default is False i.e. the environment map covers the full sphere. Note that if hemi=True, the number of REM samples in azimuth becomes n_azi $\times$ 2. This is the recommended mode (hemi=True) for MORTICIA purposes, since it reduces execution time.
- **n_sza** – The number of solar zenith angles (SZA) at which to perform transmittance and path radiance computations. Each SZA will result in another run of the base case (no radiances)

class librad.**RadEnv**(*base_case*, *n_pol*, *n_azi*, *mxumu=48*, *mxphi=19*, *hemi=False*, *n_sza=0*)

RadEnv is a class to encapsulate a large number of uvspec runs to cover a large number of sightlines over the whole sphere. A radiance map over the complete sphere is called a radiant environment map. The uvspec utility can only handle a limited number of sighlines per run, determined by the maximum number of polar and azimuthal angles specified in the file /libsrc_f/DISORT.MXD. If these values are changed, DISORT and uvspec must be recompiled. If the values are set too large, the memory requirements could easily exceed your computer's limit (there is currently no dynamic memory allocation in DISORT). The situation for the cdisort solver is less clear.

Note: The *polradtran* radiative transfer (RT) solver does not produce direct irradiance outputs (*edir*), This presents a problem for calculation of path transmission (optical depth) between output atmospheric levels (e.g. as specified by the uvspec *zout* keyword). This solver only produces total fluxes (irradiances) for each of desired stokes parameters. Hence for calculation of polarised radiant environment maps (REMs), the only feasible option for *MORTICIA* is to use the *mystic* solver with the *mc_polarisation* option. Currently, the librad.Case uvspec output file reading functions do not cater for *mystic*.

Create a set of uvspec runs covering the whole sphere to calculate a full radiant environment map. Where the base_case is the uvspec case on which to base the environmental map, Name is the name to give the environmental map and n_pol and n_azi are the number of polar and azimuthal sightline angles to generate. The mxumu and mxphi are the maximum number of polar and azimuth angles to calculate in a single run of uvspec. The default values are mxumu = 48, and mxphi = 19. These values are taken from the standard libRadtran distribution (/libsrc_f/DISORT.MXD) maximum parameter file. If using the polradtran solver, the corresponding file is /libsrc_f/POLRADTRAN.MXD. Other solvers may have different restrictions. A warning will be issued if the solver is not in the DISORT/POLRADTRAN family.

Parameters

- **base_case** – librad.Case object providing the case on which the environment map is to be based. Note that not any basecase can be used. As a general guideline, the basecase should have standard irradiance outputs (i.e. should not use the *output_user* keyword). It should also not the use *output_process* or *output_quantity* keywords, which change the units and/or format of the libRadtran/uvspec output.

  Minimal validation of the basecase is performed. However, use with *mol_abs_param* such as *kato* and *fu* is important for *MORTICIA* and these are supported (k-distribution or *correlated-k* parametrizations). Use of *output_process*

per_nm is appropriate for *source thermal* REMs to get radiance units per nanometre rather than per inverse cm.

- **n_pol** – Number of polar angles (view/propagation zenith angles)
- **n_azi** – Number of azimuthal angles.
- **mxumu** – Maximum number of polar angles per case.
- **mxphi** – Maximum number of azimuthal angles per case.
- **hemi** – If set True, will generate only a single hemisphere being on one side of the solar principle plane. Default is False i.e. the environment map covers the full sphere. Note that if hemi=True, the number of REM samples in azimuth becomes n_azi $\times$ 2. This is the recommended mode (hemi=True) for MORTICIA purposes, since it reduces execution time.
- **n_sza** – The number of solar zenith angles (SZA) at which to perform transmittance and path radiance computations. Each SZA will result in another run of the base case (no radiances)

The solver cdisort may have dynamic memory allocation, so the warning is still issued because the situation is less clear.

A note about radiative propagation angles and viewing angles, which are 180 deg opposite to each other. For radiance calculations define the cosine of the viewing zenith angle *umu* and the sensor azimuth *phi* and don't forget to also specify the solar azimuth *phi0*. *umu* > 0 means sensor looking downward (e.g. a satellite), *umu* < 0 means looking upward. *phi* = *phi0* indicates that the sensor looks into the direction of the sun, *phi* - *phi0* = 180 means that the sun is in the back of the sensor.

phi is the propagation azimuth angle *paz*, except that *paz* is in radians and *phi* is in degrees.

pza is the propagation zenith angle in radians.

vaz is the view azimuth angle and is 180 ° different from *paz*. *vaz* is expressed in degrees. *vza*, the view zenith angle is 180 ° different from *paz' and is expressed in degrees. In order to keep all azimuth angles in increasing order, 'vaz' is in the range [-180, 180], while 'phi* is in the range [0, 360] and *vaz* = *phi* - 180.

The value of *umu* is the cosine of the propagation zenith angle. The value of *phi* is the true azimuth of the propagation direction.

For all one-dimensional solvers the absolute azimuth does not matter, but only the relative azimuth *phi* - *phi0*.

For *MORTICIA* work, it is strongly recommended that the *hemi* flag be set True. This will automatically set the *phi0* keyword to zero in the RadEnv cases when running uvspec. This will essentially halve the execution time for radiant environment maps of the same effective spatial resolution.

__init__ (*base_case*, *n_pol*, *n_azi*, *mxumu=48*, *mxphi=19*, *hemi=False*, *n_sza=0*)

Create a set of uvspec runs covering the whole sphere to calculate a full radiant environment map. Where the base_case is the uvspec case on which to base the environmental map, Name is the name to give the environmental map and n_pol and n_azi are the number of polar and azimuthal sightline angles to generate. The mxumu and mxphi are the maximum number of polar and azimuth angles to calculate in a single run of uvspec. The default values are mxumu = 48, and mxphi = 19. These values are taken from the standard libRadtran distribution (/libsrc_f/DISORT.MXD) maximum parameter file. If using the polradtran solver, the corresponding file is /libsrc_f/POLRADTRAN.MXD. Other solvers may have different restrictions. A warning will be issued if the solver is not in the DISORT/POLRADTRAN family.

Parameters

- **base_case** – librad.Case object providing the case on which the environment map is to be based. Note that not any basecase can be used. As a general guideline, the basecase should have standard irradiance outputs (i.e. should not use the *output_user* keyword). It should also not the use *output_process* or *output_quantity* keywords, which change the units and/or format of the libRadtran/uvspec output.

Minimal validation of the basecase is performed. However, use with *mol_abs_param* such as *kato* and *fu* is important for *MORTICIA* and these are supported (k-distribution or *correlated-k* parametrizations). Use of *output_process per_nm* is appropriate for *source thermal* REMs to get radiance units per nanometre rather than per inverse cm.

- **n_pol** – Number of polar angles (view/propagation zenith angles)
- **n_azi** – Number of azimuthal angles.
- **mxumu** – Maximum number of polar angles per case.
- **mxphi** – Maximum number of azimuthal angles per case.
- **hemi** – If set True, will generate only a single hemisphere being on one side of the solar principle plane. Default is False i.e. the environment map covers the full sphere. Note that if hemi=True, the number of REM samples in azimuth becomes n_azi $\times$ 2. This is the recommended mode (hemi=True) for MORTICIA purposes, since it reduces execution time.
- **n_sza** – The number of solar zenith angles (SZA) at which to perform transmittance and path radiance computations. Each SZA will result in another run of the base case (no radiances)

The solver cdisort may have dynamic memory allocation, so the warning is still issued because the situation is less clear.

A note about radiative propagation angles and viewing angles, which are 180 deg opposite to each other. For radiance calculations define the cosine of the viewing zenith angle *umu* and the sensor azimuth *phi* and don't forget to also specify the solar azimuth *phi0*. *umu* > 0 means sensor looking downward (e.g. a satellite), *umu* < 0 means looking upward. *phi* = *phi0* indicates that the sensor looks into the direction of the sun, *phi* - *phi0* = 180 means that the sun is in the back of the sensor.

phi is the propagation azimuth angle *paz*, except that *paz* is in radians and *phi* is in degrees.

pza is the propagation zenith angle in radians.

vaz is the view azimuth angle and is 180 ° different from *paz*. *vaz* is expressed in degrees. *vza*, the view zenith angle is 180 ° different from *paz' and is expressed in degrees. In order to keep all azimuth angles in increasing order, 'vaz' is in the range [-180, 180], while 'phi* is in the range [0, 360] and *vaz* = *phi* - 180.

The value of *umu* is the cosine of the propagation zenith angle. The value of *phi* is the true azimuth of the propagation direction.

For all one-dimensional solvers the absolute azimuth does not matter, but only the relative azimuth *phi* - *phi0*.

For *MORTICIA* work, it is strongly recommended that the *hemi* flag be set True. This will automatically set the *phi0* keyword to zero in the RadEnv cases when running uvspec. This will essentially halve the execution time for radiant environment maps of the same effective spatial resolution.

compute_path_radiance()

Compute path radiances for path segments between all altitudes in the REM. The path transmittances (optical depth) as well as the total radiances at each altitude are required to calculate path radiances. If $L^{\downarrow}_{pi}$ is the downwelling path radiance at level i originating between level i and level $i+1$ and $L^{\uparrow}_{pi}$ is the upwelling path radiance at level i originating between level i and level $i-1$, then

$$L^{\downarrow}_{pi} = L^{\downarrow}_{i} - L^{\downarrow}_{i+1}\tau^{\downarrow}_{i+1},$$

and

$$L^{\uparrow}_{pi} = L^{\uparrow}_{i} - L^{\uparrow}_{i-1}\tau^{\uparrow}_{i-1}.$$

$L^{\uparrow}_{i}$ is the lower hemisphere (upwelling hemisphere) of the REM and $\tau^{\uparrow}_{i}$ is the transmission between level i and level $i+1$.

See also:

RadEnv.compute_path_transmittance()

Returns None

compute_path_transmittance()
Compute path transmittances from the set of libRadtran/uvspec runs executed for solar zenith angles of 0 to near 90 degrees.

This method computes optical depth (-log(transmittance)) of all paths from a particular level, both upward and downward. Paths that lie in the horizontal "blind zone" are assigned OD of 0.0. These should actually be assigned OD of np.nan or perhaps np.inf.

See also:

RadEnv.setup_trans_cases()

Returns None

run_ipyparallel(*ipyparallel_view*, *stderr_to_file=False*, *purge=False*)
Run a complete set of radiant environment map cases of libRadtran/uvspec using the *ipyparallel* Python package, which provides parallel computation from Jupyter notebooks and other Python launch modes. Typical code for setting up the view:

```
from ipyparallel import Client
paraclient = Client(profile='mycluster', sshserver='me@mycluster.info', password='myp
paraclient[:].use_dill()  # Need dill as a pickle replacement for our purposes here
ipyparallel_view = paraclient.load_balanced_view()
ipyparallel_view.block = True  # Must wait for completion of all tasks on the cluster
```

Note that if new ipengines are started, use_dill() must be executed again. The use_dill() call should be a routine before every function map to the cluster.

Parameters

- **ipyparallel_view** – an ipyparallel view of a Python engine cluster (see ipyparallel documentation.)
- **stderr_to_file** – If set to True, standard error output will be sent to a file. use only for debugging purposes.
- **purge** – Boolean. If set True, the actual libRadtran cases that are executed to make up the REM are deleted in order to reduce the size of the object. If the object is purged, it is not possible to rerun the REM. Default is False - no purging (or minimal purging) is performed.

Returns

run_parallel(*n_nodes=4*)
Run the RadEnv in multiprocessing mode on the local host.

This is not yet tested, but should work with the multiprocessing package on the local host to use all the available cores. Will only work if libRadtran is installed on the local host.

In order to use dill instead of pickle, it is necessary to use the pathos multiprocessing module instead of the standard multiprocessing module

Parameters **n_nodes** – Number of compute nodes to use. Default is 4. Preferably set to number of cores you have available on the local host.

Returns

setup_trans_cases(*n_sza=32*)
Setup a list of cases for computing the transmission matrices between every level defined in the REM

(and at every wavelength and stokes parameter combination). The computation of transmittance between levels is accomplished in *MORTICIA* using libRadtran/uvspec by computing the direct solar irradiance transmittance for multiple zenith angles.

Note that if there is an optically thick cloud layer between two levels in the REM, the transmittance will compute as zero or very small. This will also result in the incorrect/noisy computation of path radiances between the two levels. This situation is unavoidable. The recommended approach is that REMS be computed with altitude levels that all lie between cloud layers (i.e. no levels in the REM span a cloud layer). The user, or the code which uses the REMs should see to this. Essentially it must be recognised that optical surveillance is not possible through optically thick cloud layers.

The REM is provided with a cloud flag that indicates if the base case incorporates clouds. The approach to computing inter-level transmittance (optical depth is the stored parameter, since this scales more closely in a linear fashion with distance) in the presence of cloud is to vary the 'cloudcover' keyword parameter (water and ice clouds independently). This is done for solar zenith angle of zero. The optical thickness from TOA to the level in question is then computed as a function of cloud cover fraction (CCF). The optical depth between levels (i.e the optical depth of a layer between two levels) is computed as the difference in optical depth to TOA of the lower level minus the optical depth to TOA of the upper level. If there is no difference in the layer optical depth when the CCF is varied from zero to some positive value (say 0.1, but not as high as 1), then the layer is free of cloud.

A flag per layer is thus obtained which indicates if the layer contains cloud. If it does, the transmittance will compute as zero between the two levels in question. This means that the cloud base is not resolved to better than the level resolution in the REM. It is probably then quite important to ensure that cloud base altitude statistics are available in the theatre climatology.

An upgrade to cloud handling could be to read the cloud profile files in order to obtain the exact vertical location of the cloud layers.

Another inherent and unavoidable problem with computation of path optical depths and radiances using libRadtran/uvspec is that precisely horizontal paths cannot be dealt with using one-dimensional RT solvers. Therefore in this case, the maximum range that can be dealt with depends on the height difference between the REM levels and the maximum solar zenith angle (SZA) used for computation of optical depth.

A further implication of the above point is that path transmittances and path radiances cannot be interpolated between the SZA nearest the horizon and the horizon proper. Some form of logarithmic extrapolation could be performed, but could result in large errors due to failure of Beer's Law and other problems.

Execution and attribution of transmittance cases will provide each level with transmittance to the level above that altitude level. Therefore the topmost level will have transmittances to TOA, but the bottom level will not have transmittances (optical depths) to BOA, unless the bottom level *is* BOA. It is recommended that all MORTICIA base cases for REM include BOA as an output level.

Parameters **n_sza** – The number of solar zenith angles at which to compute path optical depth and radiances. the SZA values are computed equi-spaced in the cosine of the solar zenith angle rather that the SZA itself. This is to help with the problem that the slant range between levels increases in linear relation to the secant of the view zenith angle. The optical depths are later interpolated to the same

Returns

sph_harm_fat (*degree*)
This code was used for debugging purposes - ignore :param degree: :return:

sph_harm_fit (*degree*, *method='trapz'*)
Fit spherical harmonics to the radiant environment map (REM). One set of coefficients per wavelength or spectral channel will be fitted. The coefficients for each spectral bin/channel comprise one coefficient for order $-m$ to $+m$ for $m = 0, 1, 2, ..., n$. The total number of coefficients is $(n + 1)^2$.

The convention used for the spherical harmonics is that of Sloan with the Ramamoorthi and Hanrahan

normalization. This is a real-valued basis defined as follows:

$$y_n^m = \begin{cases} (-1)^m\sqrt{2}\Re(Y_n^m) & m > 0 \\ (-1)^m\sqrt{2}\Im(Y_n^m) & m < 0 \\ Y_n^0 & m = 0 \end{cases} = \begin{cases} (-1)^m\sqrt{2}\cos m\phi\, P_n^m(\cos\theta) & m > 0 \\ (-1)^m\sqrt{2}\sin|m|\phi\, P_n^{|m|}(\cos\theta) & m < 0 \\ K_n^0 P_n^0(\cos\theta) & m = 0 \end{cases},$$

where the complex basis functions Y_n^m are defined as:

$$Y_n^m(\theta,\phi) = K_n^m e^{im\phi} P_n^{|m|}(\cos\theta),\ n \in \mathbf{N},\ -n \le m \le n,$$

having the normalisation factor of:

$$K_n^m = \sqrt{\frac{(2n+1)(n-|m|)!}{4\pi(n+|m|)!}}.$$

The definition of the complex basis functions is consistent with the scipy.special definition of the spherical harmonics. Therefore, for fitting of the Sloan/Ramamoorthi/Hanrahan basis, the first definition is used above, that is y_n^m can be calculated from the *scipy.special* function Y_n^m as:

$$y_n^m = \begin{cases} (-1)^m\sqrt{2}\Re(Y_n^m) & m > 0 \\ (-1)^m\sqrt{2}\Im(Y_n^m) & m < 0 \\ Y_n^0 & m = 0 \end{cases}.$$

The fitted coefficients of the spherical harmonics are computed by multiplying the REM by each of the harmonics and performing double numerical integration over zenith and azimuth angle as:

$$L_n^m = \int_{\theta=0}^{\pi}\int_{\phi=0}^{2\pi} L(\theta,\phi) y_n^m(\theta,\phi)\sin\theta d\theta d\phi,$$

where L_n^m are the coefficients and $L(\theta,\phi)$ is the REM.

Parameters

- **degree** – Spherical harmonics up to this degree n, for all orders m will be fitted.
- **method** – Integration method by which the coefficients are computed. 'trapz' for trapezoidal integration, 'sum' for simple summation and 'simpson' for Simpson's Rule. The 'trapz' method seems to be considerably more accurate than 'sum' or 'simpson'. Therefore 'trapz' is the default.

Returns

write_openexr(*filename*, *chan_names=None*, *chan_per_exr=3*, *normalise=False*, *half=False*, *repeat_azi=1*, *use_mitsuba_wvl=False*)

Write a radiant environment as an OpenEXR file or set of OpenEXR files.

Use of this method requires that the OpenEXR package be installed on your platform. This can be a problem for Windows. However, it should be possible to find a Python wheel (.whl) file for Windows which will enable installation if pip cannot do the job.

Parameters

- **filename** – Name of the file to which the OpenEXR environment map should be written. If multiple exr files are written, this will be the filename prefix. No default.
- **chan_names** – String or list of strings giving the names to be used for the channels. For example give 'RGB' if channels 'R', 'G' and 'B' if RGB files are to be written. This is the same as giving ['R', 'G', 'B']. If there are insufficient channel names for all the wavelengths, the names are reused.
- **chan_per_exr** – Number of spectral channels to write per exr file. Default is 3.
- **normalise** – Boolean. Normalise all channels to the maximum value (in any one file). Default False. Normalisation will make ti possible to display the .exr file in IrfanView or mtsgui (Mitsuba GUI), but will destroy the radiometric correctness.

- **half** – Boolean. Use float16 instead of default float32. Halves data size at cost of radiometric resolution.
- **repeat_azi** – Integer. Repeat the data in the REM azimuthal direction. This is mostly a convenience for display purposes when the REM has no azimuthal variation (e.g. in the thermal bands).
- **use_mitsuba_wvl** – Boolean. If set True, write channels using the Mitsuba wavelength assignments i.e. remap wavelengths to the 360 nm to 830 nm range. Setting this flag True will also scale the radiance values to W/m^2/sr/nm, which are the canonical units for Mitsuba rendering in the context of MORTICIA.

Polarization is not dealt with. Only the first Stokes component (I) is written to the EXR files. The number of channel names should be equal to the number of channels per EXR file, but this is not enforced. Irfanview can display three-component, normalised EXR files only. The Mitsuba GUI (mtsgui) can display EXR files with any number of channels, but it is necessary to step through the channels (using [and]) and they are displayed in grayscale. Even mtsgui will clip EXR files having radiance values exceeding 1.0.

Returns

librad.**angstrom_law**(*wavelength*, *alpha*, *beta*)

Calculation of aerosol optical thickness according to the Angstrom law. The Angstrom law is a simple power law that typifies aerosol optical thickness (aka optical depth) variation with wavelength. The Angstrom law is expressed as .. math:

```
\tau_{aer}=\beta\lambda^{-\alpha}
```

The Angstrom law can be used to set aerosol optical thickness in libRadtran/uvspec using the *aerosol_angstrom* keyword.

Parameters

- **wavelength** – Wavelength(s) at which to compute the aerosol optical thickness. If any of the wavelengths are greater than 100, then wavelengths are assumed to be in nm, otherwise wavelengths are assumed to be in microns.
- **alpha** – The Angstrom alpha exponent.
- **beta** – The Angstrom beta (aerosol optical thickness at a wavelength of 1000 nm) parameter

Returns Aerosol optical thickness at given wavelengths with Angstrom alpha and beta parameters as provided.

See also:

librad.angstrom_law_fit, the libRadtran manual

librad.**angstrom_law_fit**(*wavelength*, *aot*)

Uses scipy.optimize to fit the Angstrom power law to an array of aerosol optical thickness values given at an array of wavelengths.

Parameters

- **wavelength** – Wavelengths at which the aerosol optical thickness (aka optical depth) is provided in the aot input. If any of the wavelengths is larger than 100, then wavelengths are assumed to be in nm, otherwise wavelengths are assumed to be in microns.
- **aot** – Aerosol optical thickness at the given wavelengths

Returns Angstrom alpha and beta parameters that best fit the input AOT data in the least squares sense.

See also:

librad.angstrom_law

librad.**king_byrne_formula**(*wavelength*, *alpha_0*, *alpha_1*, *alpha_2*)

The King Byrne formula for aerosol optical depth variation with wavelength. The King Byrne formula is

$$\tau_{aer} = e^{\alpha_0} \lambda^{\alpha_1} \lambda^{-\alpha_2}$$

Parameters

- **wavelength** – Wavelengths at which the aerosol optical thickness (aka optical depth) is provided in the aot input. If any of the wavelengths is larger than 100, then wavelengths are assumed to be in nm, otherwise wavelengths are assumed to be in microns.
- **alpha_0** – The α_0 parameter
- **alpha_1** – The α_1 parameter
- **alpha_2** – The α_2 parameter

Returns Aerosol optical thickness at given wavelengths calculated with the King Byrne formula

librad.**king_byrne_formula_fit**(*wavelength*, *aot*)

Uses scipy.optimize to fit the King Byrne formula to an array of aerosol optical thickness values provided at the given wavelengths

Parameters

- **wavelength** – Wavelengths at which the aerosol optical thickness (aka optical depth) is provided in the aot input. If any of the wavelengths is larger than 100, then wavelengths are assumed to be in nm, otherwise wavelengths are assumed to be in microns.
- **aot** – Aerosol optical thickness at the given wavelengths

Returns The α_0, α_1 and α_2

librad.**koschmieder_vis**(*ext_550=None*, *aot_550=None*, *scale_height=None*, *rayleigh_ext=0.01159*)

Compute visibility in km using the Koschmieder relationship

$$V_K = \frac{\ln 50}{\epsilon_{aer} + \epsilon_{ray}}$$

Or,

$$V_K = \frac{\ln 50}{\tau_{aer}/H + \epsilon_{ray}}$$

where V_K is the visibility in km, ϵ_{aer} is the aerosol extinction coefficient at 550 nm in units of inverse km, τ_{aer} is the vertical aerosol optical depth at 550 nm, and ϵ_{aer} is the Rayleigh extinction coefficient, in units of inverse km.

H is the scale height or effective mixing layer height assuming that all aerosols are in the mixing layer. The boundary layer height is usually a good approximation.

Provide either `ext_550` OR `aot_550` together with `scale_height`. The Rayleigh extinction coefficient is optional.

Parameters

- **ext_550** – aerosol extinction coefficient τ_{aer} at 550nm in units of inverse km
- **aot_550** – vertical aerosol optical thickness
- **scale_height** – effective boundary layer height (ABL/mixing height if all aerosols in the mixing layer) in km
- **rayleigh_ext** – Rayleigh extinction coefficient, defaults to 0.01159 per km.

Returns Visibility in km according to the Koschmieder relationship

The relationship used here is taken from the MODTRAN manual.

librad.**lookup_nearest_in_file**(*filename*, *values_and_offsets*, *column_number=0*)
Look up the nearest value in a free form text file of numeric data

Parameters

- **filename** – File in which to look up the values
- **column_number** – Column number in which to search, starting from 0 - the default is 0
- **values_and_offsets** – An array with the values to search for (nominal values) in the first column and the minimum offsets to allow from the nominal values. Negative offsets will return the largest value which is less than the nominal value by at least the absolute value of the offset. Postitive offsets will return the value which is greater than the nominal value by at least the absolute offset value.

Returns List of lookup values which are nearest to the nominal values by at least the offset values.

In the tradition of libRadtran data files, lines starting with # are considered to be comments. The data in the column is assumed to be in monotonic, increasing order. Very large files should probably not be the subject of this function.

The following excample will fetch the Thuillier solar spectrum wavelengths that span the range of 385 nm to 955 nm with a margin of 2 nm on either side. This is useful when setting the uspec 'wavelength' keyword, which must give wavelengths that are actually listed in the source solar sepctrum file. >>> import morticia.rad.librad as librad >>> wavelengths = librad.lookup_nearest_in_file('data/Solar_irradiance_Thuillier_2002.txt', [[385.0, -2.0], [955, 2.0]])

5.2.2 Radiometric Utilities Module

The radiometric utilitties module, *radute* contains utility code for handling of radiometric quantities. This includes a class *Flt* for reading, writing, creating and plotting of MODTRAN-style .flt (spectral response/filter function) files.

class radute.**Flt**(*name, units='nm', filterheaders=[], filters=[], centers=[], fwhms=[], shapes=['gauss'], yedges=[0.001], centerflats=[0.0], peakvals=[1.0], wvmins=[], wvmaxs=[], oobs=[4.9406564584124654e-324]*)
Encapsulates a MODTRAN-style .flt spectral response function/filter function definition file.

Create a filter definition object (MODTRAN flt style) Input name is mandatory. All other inputs are optional, but the filterheaders list must have the same number of string elements as the number of filters. Also, either the filters are given explicitly in the filters input, or a list of filter definitions are provided for use with rad.srfgen(). If not empty, inputs centers through to oobs must be either scalar or have the same number of list elements as the filterheaders list. If scalar, the value will be replicated up to the number of filterheader values.

Parameters

- **name** – Name of the set of filters. If the filterheaders input to this constructor function is empty, an attempt will be made to read the data from a file called name + '.flt'
- **units** – Spectral coordinate units for the filter, 'nm', 'um' or 'cm^-1'
- **filterheaders** – List of strings, one header for each filter/SRF in the set.
- **filters** – A list of numpy arrays. Each list element must comprise a 2-column numpy array with the spectral coordinate (wavelength in nm or micron or wavenumber per cm) in the first column and the filter magnitude in the second column.
- **centres** – rather than provide filters, the inputs to rad.srfgen can be provided, this is a list of center wavelengths in nm
- **fwhms** – list of full width at half maximum in nm

- **shapes** – list of strings providing the shapes of the filters (see rad.srfgen for alternatives)
- **yedges** – list of yedge values (see rad.srfgen)
- **centerflats** – list of centerflat values (see rad.srfgen). Note that giving a centerflat value adds this amount ot the fwhm of the filters (broadens the resulting width to centerflat + fwhm)
- **peakvals** – list of peak values of the filters
- **wvmins** – list of minimum wavelength limits in nm
- **wvmaxs** – list of maximum wavelength limits in nm
- **oobs** – list of out-of-band leakage values

Returns MODTRAN-style filter/SRF object

__init__(*name, units='nm', filterheaders=[], filters=[], centers=[], fwhms=[], shapes=['gauss'], yedges=[0.001], centerflats=[0.0], peakvals=[1.0], wvmins=[], wvmaxs=[], oobs=[4.9406564584124654e-324]*)

Create a filter definition object (MODTRAN flt style) Input name is mandatory. All other inputs are optional, but the filterheaders list must have the same number of string elements as the number of filters. Also, either the filters are given explicitly in the filters input, or a list of filter definitions are provided for use with rad.srfgen(). If not empty, inputs centers through to oobs must be either scalar or have the same number of list elements as the filterheaders list. If scalar, the value will be replicated up to the number of filterheader values.

Parameters

- **name** – Name of the set of filters. If the filterheaders input to this constructor function is empty, an attempt will be made to read the data from a file called name + '.flt'
- **units** – Spectral coordinate units for the filter, 'nm', 'um' or 'cm^-1'
- **filterheaders** – List of strings, one header for each filter/SRF in the set.
- **filters** – A list of numpy arrays. Each list element must comprise a 2-column numpy array with the spectral coordinate (wavelength in nm or micron or wavenumber per cm) in the first column and the filter magnitude in the second column.
- **centres** – rather than provide filters, the inputs to rad.srfgen can be provided, this is a list of center wavelengths in nm
- **fwhms** – list of full width at half maximum in nm
- **shapes** – list of strings providing the shapes of the filters (see rad.srfgen for alternatives)
- **yedges** – list of yedge values (see rad.srfgen)
- **centerflats** – list of centerflat values (see rad.srfgen). Note that giving a centerflat value adds this amount ot the fwhm of the filters (broadens the resulting width to centerflat + fwhm)
- **peakvals** – list of peak values of the filters
- **wvmins** – list of minimum wavelength limits in nm
- **wvmaxs** – list of maximum wavelength limits in nm
- **oobs** – list of out-of-band leakage values

Returns MODTRAN-style filter/SRF object

static checkparm(*parmname*, *parm*, *nfilters*)

Input parameter checking for Flt constructor

Parameters

- **parmname** – Name of parameter fpr checking
- **parm** – Parameter value
- **nfilters** – Number of filters

Returns Checked parameter

flt_as_xd()
Convert an Flt class object to a list of xarray DataArray objects

Returns The set of Flt filters as a list of xarray DataArray objects, with a wavelength coordinate axis ('wvl', long_name = 'Wavelength'

flt_as_xd_harmonised(*quantity_name='srf'*, *chn_start_index=0*)
Convert the Flt class object into a single, wavelength-harmonised xr.DataArray.

Parameters

- **quantity_name** – The name of the quantity as defined the long_names variable in moglo.py. Defaults to 'srf', a Spectral Response Function, but could also be a transmission functions 'trn' or other spectral quantity known to moglo.py.
- **chn_start_index** – Use this parameter to select the starting channel number. Defaults to zero.

Returns The set of Flt filters as a single, wavelength-harmonised xr.DataArray object. The filter headers are returned in an attribute called 'labels'. The fileheader of the Flt object is returned in an attribute called 'title' (netCDF recommendation)

plot(*filter_numbers=None*)
Plot a MODTRAN-style set of filter/SRF curves.

Parameters **filter_numbers** – List of filter numbers to plot, defaults to all filters. Filter indices start at 0.

Returns None

read(*filename*, *name='Unknown'*)
Read a .flt format spectral band filter definitions file (MODTRAN format)

Parameters **filename** –

Returns object of class Flt, if the file is a well-formatted MODTRAN-style .flt file

write(*filename=None*, *format=' %f'*)
Write a MODTRAN-style .flt file for this filter/SRF set

Parameters

- **filename** – Optional filename without extension. If not given, the filter name will be used with
- **format** – Format specifier as for np.savetext for writing the data, default is ' %f' extension .flt

Returns None

class radute.**SpectralDistribution**(*extreme_limits*, *in_band_limits=None*, *in_band_threshold=0.01*)
The SpectralDistribution class defines any band-limited spectral distribution function. This could be the spectral response functions of a sensor, or the spectral transmittance of an optical filter, the spectral radiance, irradiance or any other band-limited spectral quantity.

Create a basic, spectral distribution function with certain spectral band limits

By default, wavelenths are specified in nm for SpectralDistribution and SpectralSpace objects.

Returns A SpectralDistribution object

__init__ (*extreme_limits*, *in_band_limits=None*, *in_band_threshold=0.01*)
Create a basic, spectral distribution function with certain spectral band limits

By default, wavelenths are specified in nm for SpectralDistribution and SpectralSpace objects.

Returns A SpectralDistribution object

classmethod avhrr_kratz (*i_channel*, *resolution=0.001*)

Parameters

- **i_channel** – The single AVHRR Kratz channel to obtain. Integer 1 to 16
- **resolution** – this is the spectral resolution in nm of the band edges. Default 0.001 nm

Returns A SpectralDistribution object defining the requested avhrr_kratz channel

decimate_resolution ()
Reduce the number of sample points representing the spectral distribution in an intelligent way.

This procedure only works for positive spectral power distributions.

Returns

classmethod fu (*i_channel*, *resolution=0.001*)
Return a Fu correlated-k channel as a SpectralDistribution

Parameters

- **i_channel** – the single Fu channel to be obtained. Integer 1 to 18
- **resolution** – this is the spectral resolution in nm of the band edges. Default 0.001 nm

Returns A SpectralDistribution object defining the requested Fu channel

classmethod kato (*i_channel*, *resolution=0.001*)
Return a Kato correlated-k channel definition as a SpectralDistribution. Only a single channel can be represented. Use a SpectralSpace to get multiple Kato channels

Parameters

- **i_channel** – the single kato channel to be obtained. Integer 1 to 32
- **resolution** – this is the spectral resolution in nm of the band edges. Default 0.001 nm which is typically adequate.

Returns A SpectralDistribution object defining the requested Kato channel

classmethod sensor_channel (*platform_series*, *platform_name*, *sensor_name*, *channel_name*)
Load a spectral response file from the libRadtran-compatible library as a SpectralDistribution

The available response function filter files can be viewed in the sub-directory rad/radata/filter. Note that only a single channel can be loaded using this method. To load multiple channels, use SpectralSpace.sensor_channels().

Parameters

- **platform_series** – Name of the series of platforms on which the sensor is carried e.g. 'landsat'
- **platform_name** – Name of the specific satellite/platform on which the sensor is carried e.g. 'landsat7'
- **sensor_name** – Name of the specific sensor on the platform e.g. 'tm' or 'etm'
- **channel_name** – Name of the specific spectral channel on the given sensor e.g.

Returns

classmethod **spectral_slice**(*extreme_limits*, *in_band_limits*, *resolution=0.001*, *oob_leakage=0.0*)

Create a SpectralDistribution object as a simple spectral slice

Parameters

- **extreme_limits** –
- **in_band_limits** –
- **resolution** –
- **oob_leakage** –

Returns

class radute.**SpectralSpace**

A SpectralSpace is a set (represented as a list) of SpectralDistribution objects.

For example, the full Kato correlated-k list of spectral slices constitutes a SpectralSpace. A sub-range of of Kato or other correlated-k channels (Fu or avhrr_kratz) also qualify. The spectral response functions of a sensor can also be represented using a SpectralSpace.

An important use of SpectralSpaces is to compute "projections" which could also be thought of as "dot products". The projection of one SpectralSpace into another comprises multplying the distribution functions and integrating over wavelength to obtain a set of weights. The integral is typically also normalised to retain equivalent units.

Here are some examples: A set of spectral end-member functions can be represented as a SpectralSpace. These end-members might be propagated to calculate an end-member response at a camera focal plane. The end-members are projected onto the spectral response functions of the sensor to get the sensor responses to each of the end-members.

Notes: A SpectralSpace is not generally orthogonal unless specifically designed so by the user.

classmethod **from_flt_file**(*filename*, *re_select=None*)

Create a SpectralSpace list of SpectralDistribution objects by reading a MODTRAN-compatible "filter function" .flt file.

Parameters

- **filename** – The MODTRAN-compatible .flt file from which to read the filter functions.
- **re_select** – Select a sub-set of the channels using a regular expression filter. Only the channel names/descriptions that match the regular expression will be included. The default is to read all channels in the file.

Returns

radute.**readOpenEXR**(*filename*)

Simple read function for an OpenEXR file

Use of this function requires that the OpenEXR package be installed. :param filename: The name of the OpenEXR file :return channel_names: List of image channel names found in the OpenEXR file :return im_dict: All image data in a dictionary keyed by channel names or channel groups. Channels are grouped

into RGB triplets if the channel names have the form prefix.R, prefix.G and prefix.B. All image data is returned as numpy arrays.

Return header OpenEXR header as a dictionary.

radute.**srfgen**(*center*, *fwhm*, *n=101*, *shape='gauss'*, *yedge=0.001*, *wvmin=None*, *wvmax=None*, *centerflat=0.0*, *oob=4.9406564584124654e-324*, *peakval=1.0*, *units='nm'*)

Generate a spectral filter or spectral response function of various shapes

Parameters

- **center** – center wavelength of the filter in nm

- **fwhm** – full width at half maximum of the filter in nm
- **n** – number of spectral samples in the filter (minimum of 3, default 101), should be odd
- **shape** – filter shape, one of 'gauss', 'bartlett', 'welch', 'cosine', 'box', 'cos^2' (default 'gauss')
- **yedge** – minimum y-value at the limits of the filter (default 0.001). No filter values below this threshold
- **wvmin** – extend spectral definition by adding a single point at (wvmin, oob)
- **wvmax** – exptend spectral definition by adding a single point at (wvmax, oob)
- **centerflat** – opens a flat region in the centre of filter having a width of centerflat nm
- **oob** – out-of-band leakage, default 0.0, must be <= yedge
- **peakval** – simply scales the peak of the filter function to this value (default 1.0)
- **units** – spectral axis units, either 'nm', 'cm^-1' or 'um' (nanometers, wavenumber per cm or microns) will be returned

Returns wvlnm, y, wvn, wvlum (wavelengths in nm, filter values, wavenumbers per cm and wavelengths in microns)

radute.**tophat**(*center*, *fwhm*, *delta=0.0*, *wvmin=None*, *wvmax=None*, *oob=0.0*, *units='nm'*)

Return tophat/box filter defined by 6 points Can also specify out-of-band values and extreme limits, which adds upper and lower bound points

Parameters

- **center** – center wavelength in nm
- **fwhm** – full width at half max in nm
- **delta** – smallest x-coordinate increment, default see np.nextafter
- **wvmin** – minimum wavelength to reach default None
- **wvmax** – maximum wavelength to reach default None
- **oob** – out-of-band leakage
- **units** – spectral axis units, either 'nm', 'cm^-1' or 'um' (nanometers, wavenumber per cm or microns) will be returned

Returns wvlnm, y, wvn, wvlum (wavelengths in nm, filter values, wavenumbers per cm and wavelengths in microns)

5.3 Scene Package

The Scene package provides all MORTICIA functionality relating to radiometry and atmospheric radiative transfer. Atmospheric radiative transfer is performed by the libRadtran suite of tools.

5.3.1 Target Module

The Target module concerns itself with calculations for rendering a target

- Greyscale Beach Target
- Target 2
- Target 3

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

SEVEN

JUPYTER NOTEBOOKS

Jupyter notebooks are provided to illustrate usage of the main package (`morticia`) as well as sub-packages documented below.

e

o

r