pyradi Documentation

Release

pyradi team

Jul 29, 2016

The pyradi toolkit is a Python toolkit to perform optical and infrared computational radiometry (flux flow) calculations.

The toolkit is available at
https://pypi.python.org/pypi/pyradi/ (pip installation package)
https://github.com/NelisW/pyradi (latest version in the repository)

See docs at
http://nelisw.github.io/pyradi-docs/_build/html/index.html
https://raw.githubusercontent.com/NelisW/pyradi-docs/gh-pages/_build/latex/pyradi.pdf

IPython notebooks demonstrating the use of pyradi is available at
https://github.com/NelisW/ComputationalRadiometry#computational-optical-radiometry-with-pyradi

CHAPTER

ONE

INTRODUCTION

1.1 Overview

Electro-optical system design, data analysis and modelling involve a significant amount of calculation and processing. Many of these calculations are of a repetitive and general nature, suitable for including in a generic toolkit. The availability of such a toolkit facilitates and increases productivity during subsequent tool development: 'develop once and use many times'. The concept of an extendible toolkit lends itself naturally to the open-source philosophy, where the toolkit user-base develops the capability cooperatively, for mutual benefit. This paper covers the underlying philosophy to the toolkit development, brief descriptions and examples of the various tools and an overview of the electro-optical toolkit.

The pyradi toolbox can be applied towards many different applications. An example is included in the pyradi website (see the file exflamesensor.py). This example was first published in a SPIE conference paper *[SPIE8543Pyradi]*.

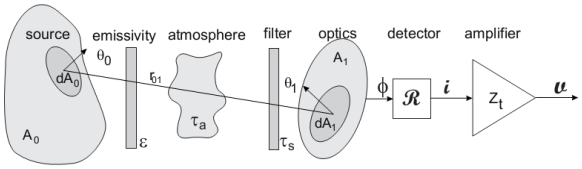
1.2 Toolkit approach

The development of this toolkit is following the Unix philosophy for software development, summarised in the words of Doug McIlroy: 'Write programs that do one thing and do it well. Write programs to work together.' In broader terms the philosophy was stated by Eric Raymond, but only selected items shown here (http://en.wikipedia.org/wiki/Unix_philosophy):

1. Rule of Modularity: Write simple parts connected by clean interfaces.
2. Rule of Clarity: Clarity is better than cleverness.
3. Rule of Composition: Design programs to be connected to other programs.
4. Rule of Simplicity: Design for simplicity; add complexity only where you must.
5. Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.
6. Rule of Transparency: Design for visibility to make inspection and debugging easier.
7. Rule of Robustness: Robustness is the child of transparency and simplicity.
8. Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.
9. Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.
10. Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.
11. Rule of Optimisation: Prototype before polishing. Get it working before you optimise it.
12. Rule of Extensibility: Design for the future, because it will be here sooner than you think.

1.3 Example application

A typical radiometry toolkit requirement (very much simplified) is the calculation of the detector current of an electro-optical sensor viewing a target object. The system can be conceptually modelled as shown in the figure below, comprising a radiating source with spectral radiance, an intervening medium (e.g. the atmosphere), a spectral filter, optics, a detector and an amplifier.

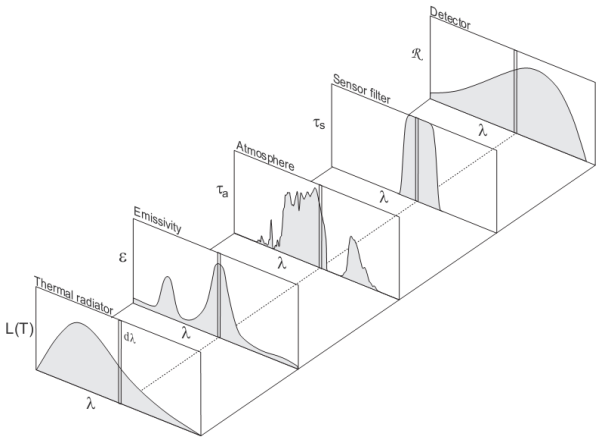


The amplifier output signal can be calculated in the following equation, by integrating over all wavelengths, over the full source area A_0 and over the optical aperture area A_1,

$$v = Z_t \int_{A_0} \int_{A_1} \frac{1}{r_{01}^2} \int_0^\infty \epsilon_\lambda L_\lambda(T, A_0) \tau_{a\lambda} \tau_{s\lambda}(A_1) \mathcal{R}_\lambda \, d\lambda \, d(\cos\theta_0 A_0) \, d(\cos\theta_1 A_1)$$

where v is the output signal voltage, r_{01} is the distance between elemental areas $d(\cos\theta_0 A_0)$ and $d(\cos\theta_1 A_1)$, ϵ_λ is the source spectral emissivity, $L_\lambda(T, A_0)$ is the Planck Law radiation at temperature T at location A_0, $\tau_{a\lambda}$ is the atmospheric spectral transmittance, $\tau_{s\lambda}(A_1)$ is the sensor spectral transmittance at location A_1, $\mathcal{R}_\lambda$ is the spectral detector responsivity in [A/W], Z_t is the amplifier transimpedance gain in [V/A]. The spectral integral $\int_0^\infty d\lambda$ accounts for the total flux for all wavelengths, the spatial integral $\int_{A_0} d(\cos\theta_0 A_0)$ accounts for flux over the total area of the source, and the spatial integral $\int_{A_1} d(\cos\theta_1 A_1)$ accounts for the total area of the receiving area.

The top graphic in the following figure illustrates the reasoning behind the spectral integral as a product, followed by an integral (summation),



$$\int_0^\infty \epsilon_\lambda L_\lambda(T) \tau_{a\lambda} \tau_{s\lambda} \mathcal{R}_\lambda \, d\lambda,$$

where the spectral variability of the source, medium and sensor parameters are multiplied as spectral variables and afterwards integrated over all wavelengths to yield the total in-band signal. The domain of spectral quantities can be stated in terms of a wavelength, wavenumber, or less often, temporal frequency.

Likewise, the source radiance is integrated over the two respective areas of the target A_0, and the sensor aperture A_1. Note that if the sensor field of view footprint at the source is smaller than the physical source area, only the flux emanating from the footprint area is integrated.

This example is a relatively complete worked example. The objective is to calculate the signal of a simple sensor, detecting the presence or absence of a flame in the sensor field of view. The sensor is pointed to an area just outside a furnace smokestack, against a clear sky background. The sensor must detect a change in signal, to indicate the presence or absence of a flame.

The sensor has an aperture area of 7.8×10^{-3} m^2 and a field of view of 1×10^{-4} sr. The sensor filter spectral transmittance is shown below. The InSb detector has a peak responsivity of 2.5 A/W and normalised spectral response shown below. The preamplifier transimpedance is 10000 V/A.

The flame area is 1 m^2, the flame temperature is 1000° C, and the emissivity is shown below. The emissivity is 0.1 over most of the spectral band, due to carbon particles in the flame. At 4.3 μm there is a strong emissivity rise due to the hot carbon dioxide CO_2 in the flame.

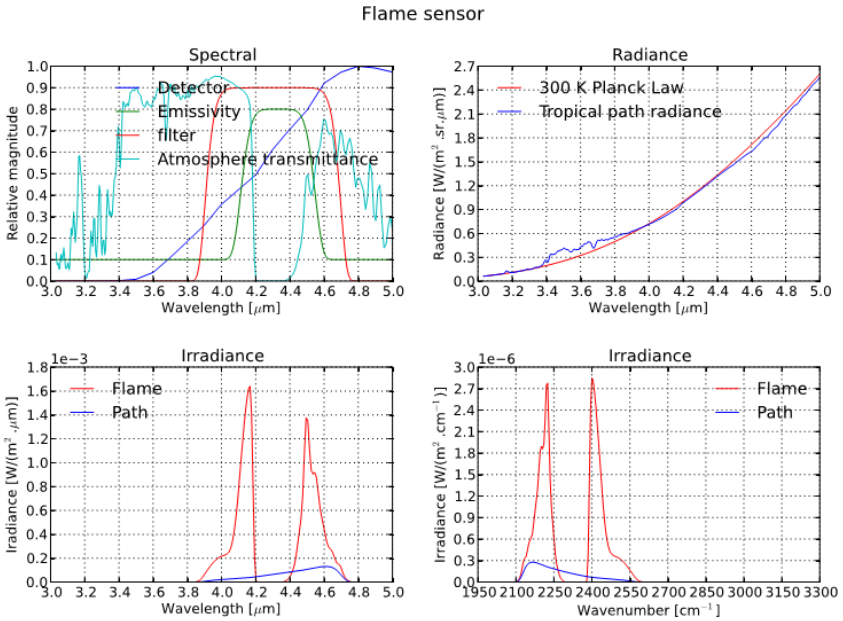
The distance between the flame and the sensor is 1000~m. The atmospheric properties are calculated with the Modtran Tropical climatic model. The path is oriented such that the sensor stares out to space, at a zenith angle of 88°. The spectral transmittance and path radiance along this path is shown in below.

The peak in the flame emissivity and the dip in atmospheric transmittance are both centered around the 4.3μm CO_2 band. The calculation of flux radiative transfer through the atmosphere must account for the strong spectral variation, by using a spectral integral.

The signal caused by the flame is given by the equation above, where the integrals over the surfaces of the flame and sensor are just their respective areas. The signal caused by the atmospheric path radiance is given by

$$v = Z_t \omega_{\text{optics}} A_{\text{optics}} \int_0^\infty L_{\text{path}\lambda} \tau_{s\lambda} \mathcal{R}_\lambda \, d\lambda,$$

where ω_{optics} is the sensor field of view, A_{optics} is the optical aperture area, $L_{\text{path}\lambda}$ is the spectral path radiance and the rest of the symbols are as defined above.



The pyradi code to model this sensor is available as exflamesensor.py. The output from this script is as follows:

```
Optics    : area=0.0078 m^2 FOV=0.0001 [sr]
Amplifier: gain=10000.0 [V/A]
Detector : peak responsivity=2.5 [A/W]
Flame     : temperature=1273.16 [K] area=1 [m^2] distance=1000 [m] fill=0.01 [-]
Flame     : irradiance= 3.29e-04 [W/m^2] signal= 0.0641 [V]
Path      : irradiance= 5.45e-05 [W/m^2] signal= 0.0106 [V]
```

It is clear that the flame signal is six times larger than the path radiance signal, even though the flame fills only 0.01 of the sensor field of view.

CHAPTER

TWO

PLANCK AND THERMAL RADIATION

2.1 Overview

This module provides functions for Planck law exitance calculations, as well as temperature derivative calculations. The functions provide spectral exitance in [W/(m^2.*)] or [q/(s.m^2.*)], given the temperature and a vector of one of wavelength, wavenumbers or frequency (six combinations each for exitance and temperature derivative). The total exitance can also be calculated by using the Stefan-Boltzman equation, in [W/m^2] or [q/(s.m^2)]. 'Exitance' is the CIE/ISO term for the older term 'emittance'.

The Planck and temperature-derivative Planck functions take the spectral variable (wavelength, wavenumber or frequency) and/or temperature as either a scalar, a single element list, a multi-element list or a numpy array.

Spectral values must be strictly scalar or shape (N,) or (N,1). Shape (1,N) will not work.

Temperature values must be strictly scalar, list[M], shape (M,), (M,1), or (1,M). Shape (Q,M) will not work.

If the spectral variable and temperature are both single numbers (scalars or lists with one element), the return value is a scalar. If either the temperature or the spectral variable are single-valued, the return value is a rank-1 vector. If both the temperature and spectral variable are multi-valued, the return value is a rank-2 array, with the spectral variable along axis=0.

This module uses the CODATA physical constants. For more details see http://physics.nist.gov/cuu/pdf/RevModPhysCODATA2010.pdf

See the __main__ function for testing and examples of use.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

2.2 Module classes

class `pyradi.ryplanck.`**`PlanckConstants`**

Precalculate the Planck function constants using the values in scipy.constants. Presumbly these constants are up to date and will be kept up to date.

This module uses the CODATA physical constants. For more details see http://physics.nist.gov/cuu/pdf/RevModPhysCODATA2010.pdf

Reference: http://docs.scipy.org/doc/scipy/reference/constants.html

`printConstants``()`

Print Planck function constants.

Args:

None

Returns:

Print to stdout

Raises:

No exception is raised.

2.3 Module functions

pyradi.ryplanck.**planck**(*spectral*, *temperature*, *type='el'*)

Planck law spectral exitance.

Calculates the Planck law spectral exitance from a surface at the stated temperature. Temperature can be a scalar, a list or an array. Exitance can be given in radiant or photon rate units, depending on user input in type.

Args:

spectral (scalar, np.array (N,) or (N,1)): spectral vector.
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]
type (string):
'e' signifies Radiant values in [W/m^2.*].
'q' signifies photon rate values [quanta/(s.m^2.*)].
'l' signifies wavelength spectral vector [micrometer].
'n' signifies wavenumber spectral vector [cm-1].
'f' signifies frequency spectral vecor [Hz].

Returns:

(scalar, np.array[N,M]): spectral radiant exitance (not radiance) in units selected.
For type = 'el' units will be [W/(m^2.um)].
For type = 'qf' units will be [q/(s.m^2.Hz)].
Other return types are similarly defined as above.
Returns None on error.

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**dplanck**(*spectral*, *temperature*, *type='el'*)

Temperature derivative of Planck law exitance.

Calculates the temperature derivative for Planck law spectral exitance from a surface at the stated temperature. dM/dT can be given in radiant or photon rate units, depending on user input in type. Temperature can be a scalar, a list or an array.

Args:

spectral (scalar, np.array (N,) or (N,1)): spectral vector in [micrometer], [cm-1] or [Hz].
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]
type (string):
'e' signifies Radiant values in [W/(m^2.K)].
'q' signifies photon rate values [quanta/(s.m^2.K)].
'l' signifies wavelength spectral vector [micrometer].
'n' signifies wavenumber spectral vector [cm-1].
'f' signifies frequency spectral vecor [Hz].

Returns:

(scalar, np.array[N,M]): spectral radiant exitance (not radiance) in units selected.
For type = 'el' units will be [W/(m2.um.K)]
For type = 'qf' units will be [q/(s.m2.Hz.K)]

Other return types are similarly defined as above.
Returns None on error.

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**stefanboltzman**(*temperature*, *type='e'*)
Stefan-Boltzman wideband integrated exitance.

Calculates the total Planck law exitance, integrated over all wavelengths, from a surface at the stated temperature. Exitance can be given in radiant or photon rate units, depending on user input in type.

Args:

(scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]
type (string): 'e' for radiant or 'q' for photon rate exitance.

Returns:

(float): integrated radiant exitance in [W/m^2] or [q/(s.m^2)].
Returns a -1 if the type is not 'e' or 'q'

Raises:

No exception is raised.

pyradi.ryplanck.**planckel**(*spectral*, *temperature*)
Planck function in wavelength for radiant exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): wavelength vector in [um]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in W/(m^2.um)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**plancken**(*spectral*, *temperature*)
Planck function in wavenumber for radiant exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): wavenumber vector in [cm^-1]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in W/(m^2.cm^-1)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**planckef**(*spectral*, *temperature*)
Planck function in frequency for radiant exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): frequency vector in [Hz]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in W/(m^2.Hz)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**planckql**(*spectral*, *temperature*)
Planck function in wavelength domain for photon rate exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): wavelength vector in [um]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in q/(s.m^2.um)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**planckqn**(*spectral*, *temperature*)
Planck function in wavenumber domain for photon rate exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): wavenumber vector in [cm^-1]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in q/(s.m^2.cm^-1)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**planckqf**(*spectral*, *temperature*)
Planck function in frequency domain for photon rate exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): frequency vector in [Hz]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in q/(s.m^2.Hz)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**dplnckel**(*spectral*, *temperature*)
Temperature derivative of Planck function in wavelength domain for radiant exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): wavelength vector in [um]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in W/(K.m^2.um)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**dplncken**(*spectral*, *temperature*)
Temperature derivative of Planck function in wavenumber domain for radiance exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): wavenumber vector in [cm^-1]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in W/(K.m^2.cm^-1)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**dplnckef**(*spectral*, *temperature*)
Temperature derivative of Planck function in frequency domain for radiant exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): frequency vector in [Hz]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance/K in W/(K.m^2.Hz)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**dplnckql**(*spectral*, *temperature*)
Temperature derivative of Planck function in wavenumber domain for radiance exitance.

Args:

spectral (scalar, np.array (N,) or (N,1)): wavelength vector in [um]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in q/(K.s.m^2.um)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**dplnckqn**(*spectral*, *temperature*)
Temperature derivative of Planck function in wavenumber domain for photon rate.

Args:

spectral (scalar, np.array (N,) or (N,1)): wavenumber vector in [cm^-1]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in q/(s.m^2.cm^-1)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**dplnckqf**(*spectral*, *temperature*)
Temperature derivative of Planck function in frequency domain for photon rate.

Args:

spectral (scalar, np.array (N,) or (N,1)): frequency vector in [Hz]
temperature (scalar, list[M], np.array (M,), (M,1) or (1,M)): Temperature in [K]

Returns:

(scalar, np.array[N,M]): spectral radiant exitance in q/(K.s.m^2.Hz)

Raises:

No exception is raised, returns None on error.

pyradi.ryplanck.**fixDimensions**(*planckFun*)

Decorator function to prepare the spectral and temperature array dimensions and order before and after the actual Planck function. The Planck functions process elementwise and therefore require flattened arrays. This decorator flattens, executes the planck function and reshape afterwards the correct shape, according to input.

CHAPTER

THREE

FILE READING/WRITING UTILITY

3.1 Overview

This module provides functions for file input/output. These are all wrapper functions, based on existing functions in other Python classes. Functions are provided to save a two-dimensional array to a text file, load selected columns of data from a text file, load a column header line, compact strings to include only legal filename characters, and a function from the Python Cookbook to recursively match filename patterns.

See the __main__ function for examples of use.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

3.2 Module functions

pyradi.ryfiles.**saveHeaderArrayTextFile**(*filename*, *dataArray*, *header=None*, *comment=None*, *delimiter=None*)

Save a numpy array to a file, included header lines.

This function saves a two-dimensional array to a text file, with an optional user-defined header. This functionality will be part of numpy 1.7, when released.

Args:

filename (string): name of the output ASCII flatfile.
dataArray (np.array[N,M]): a two-dimensional array.
header (string): the optional header.
comment (string): the symbol used to comment out lines, default value is None.
delimiter (string): delimiter used to separate columns, default is whitespace.

Returns:

Nothing.

Raises:

No exception is raised.

pyradi.ryfiles.**loadColumnTextFile**(*filename, loadCol=[1], comment=None, normalize=0, skiprows=0, delimiter=None, abscissaScale=1, ordinateScale=1, abscissaOut=None, returnAbscissa=False*)

Load selected column data from a text file, processing as specified.

This function loads column data from a text file, scaling and interpolating the read-in data, according to user specification. The first 0'th column has special significance: it is considered the abscissa (x-values) of the data set, while the remaining columns are any number of ordinate (y-value) vectors. The user passes a list of

columns to be read (default is [1]) - only these columns are read, processed and returned when the function exits.The user also passes an abscissa vector to which the input data is interpolated and then subsequently amplitude scaled or normalised.

Note: leave only single separators (e.g. spaces) between columns! Also watch out for a single space at the start of line.

Args:

filename (string): name of the input ASCII flatfile.
loadCol ([int]): the M =len([]) column(s) to be loaded as the ordinate, default value is column 1
comment (string): string, the symbol used to comment out lines, default value is None
normalize (int): integer, flag to indicate if data must be normalized.
skiprows (int): integer, the number of rows to be skipped at the start of the file (e.g. headers)
delimiter (string): string, the delimiter used to separate columns, default is whitespace.
abscissaScale (float): scale by which abscissa (column 0) must be multiplied
ordinateScale (float): scale by which ordinate (column >0) must be multiplied
abscissaOut (np.array[N,] or [N,1]): abscissa vector on which output variables are interpolated.
returnAbscissa (bool): return the abscissa vector as second item in return tuple.

Returns:

ordinatesOut (np.array[N,M]): The interpolated, M columns of N rows, processed array.
abscissaOut (np.array[N,M]): The ascissa where the ordinates are interpolated

Raises:

No exception is raised.

pyradi.ryfiles.**loadHeaderTextFile**(*filename, loadCol=[1], comment=None*)

Loads column header data in the first string of a text file.

loads column header data from a file, from the first row. Headers must be delimited by commas. The function [LoadColumnTextFile] provides more comprehensive capabilties.

Args:

filename (string): the name of the input ASCII flatfile.
loadCol ([int]): list of numbers, the column headers to be loaded , default value is column 1
comment (string): the symbol to comment out lines

Returns:

[string]: a list with selected column header entries

Raises:

No exception is raised.

pyradi.ryfiles.**cleanFilename**(*sourcestring*, *removestring=' %:/,.\\[]<>*?'*)

Clean a string by removing selected characters.

Creates a legal and 'clean' source string from a string by removing some clutter and characters not allowed in filenames. A default set is given but the user can override the default string.

Args:

sourcestring (string): the string to be cleaned.
removestring (string): remove all these characters from the string (optional).

Returns:

(string): A cleaned-up string.

Raises:

No exception is raised.

pyradi.ryfiles.**listFiles**(*root*, *patterns='*'*, *recurse=1*, *return_folders=0*, *useRegex=False*)

Lists the files/directories meeting specific requirement

Returns a list of file paths to files in a file system, searching a directory structure along the specified path, looking for files that matches the glob pattern. If specified, the search will continue into sub-directories. A list of matching names is returned. The function supports a local or network reachable filesystem, but not URLs.

Args:

root (string): directory root from where the search must take place

patterns (string): glob/regex pattern for filename matching

recurse (unt): flag to indicate if subdirectories must also be searched (optional)

return_folders (int): flag to indicate if folder names must also be returned (optional)

useRegex (bool): flag to indicate if patterns areregular expression strings (optional)

Returns:

A list with matching file/directory names

Raises:

No exception is raised.

pyradi.ryfiles.**readRawFrames**(*fname*, *rows*, *cols*, *vartype*, *loadFrames=[]*)

Loading multi-frame two-dimensional arrays from a raw data file of known data type.

The file must consist of multiple frames, all with the same number of rows and columns. Frames of different data types can be read, according to the user specification. The user can specify which frames must be loaded (if not the whole file).

Args:

fname (string): filename

rows (int): number of rows in each frame

cols (int): number of columns in each frame

vartype (np.dtype): numpy data type of data to be read

int8, int16, int32, int64

uint8, uint16, uint32, uint64

float16, float32, float64

loadFrames ([int]): optional list of frames to load, zero-based , empty list (default) loads all frames

Returns:

frames (int) : number of frames in the returned data set,

0 if error occurred

rawShaped (np.ndarray): vartype numpy array of dimensions (frames,rows,cols),

None if error occurred

Raises:

No exception is raised.

pyradi.ryfiles.**rawFrameToImageFile**(*image*, *filename*)

Writes a single raw image frame to image file. The file type must be given, e.g. png or jpg. The image need not be scaled beforehand, it is done prior to writing out the image. Could be one of BMP, JPG, JPEG, PNG, PPM, TIFF, XBM, XPM) but the file types available depends on the QT imsave plugin in use.

Args:

image (np.ndarray): two-dimensional array representing an image

filename (string): name of file to be written to, with extension

Returns:

Nothing

Raises:

No exception is raised.

pyradi.ryfiles.**arrayToLaTex**(*filename*, *arr*, *header=None*, *leftCol=None*, *formatstring='%1.4e'*, *filemode='wt'*)

Write a numpy array to latex table format in output file.

The table can contain only the array data (no top header or left column side-header), or you can add either or both of the top row or side column headers. Leave 'header' or 'leftcol' as None is you don't want these.

The output format of the array data can be specified, i.e. scientific notation or fixed decimal point.

Args:

fname (string): text writing output path and filename

arr (np.array[N,M]): array with table data

header (string): column header in final latex format (optional)

leftCol ([string]): left column each row, in final latex format (optional)

formatstring (string): output format precision for array data (see np.savetxt) (optional)

filemode (string): file open mode (a=append, w=new file) (optional)

Returns:

None, writes a file to disk

Raises:

No exception is raised.

pyradi.ryfiles.**epsLaTexFigure**(*filename*, *epsname*, *caption*, *scale=None*, *vscale=None*, *filemode='a'*, *strPost=''*)

Write the LaTeX code to include an eps graphic as a latex figure. The text is added to an existing file.

Args:

fname (string): text writing output path and filename.

epsname (string): filename/path to eps file (relative to where the LaTeX document is built).

caption (string): figure caption

scale (double): figure scale to textwidth [0..1]

vscale (double): figure scale to textheight [0..1]

filemode (string): file open mode (a=append, w=new file) (optional)

strPost (string): string to write to file after latex figure block (optional)

Returns:

None, writes a file to disk

Raises:

No exception is raised.

pyradi.ryfiles.**read2DLookupTable**(*filename*)

Read a 2D lookup table and extract the data.

The table has the following format:

```
line 1: xlabel ylabel title
line 2: 0 (vector of y (col) abscissa)
```

```
        lines 3 and following: (element of x (row) abscissa), followed
        by table data.
```

From line/row 3 onwards the first element is the x abscissa value followed by the row of data, one point for each y abscissa value.

The file format can depicted as follows:

```
        x-name y-name ordinates-name
        0 y1 y2 y3 y4
        x1 v11 v12 v13 v14
        x2 v21 v22 v23 v24
        x3 v31 v32 v33 v34
        x4 v41 v42 v43 v44
        x5 v51 v52 v53 v54
        x6 v61 v62 v63 v64
```

This function reads the file and returns the individual data items.

Args:

fname (string): input path and filename

Returns:

xVec ((np.array[N])): x abscissae
yVec ((np.array[M])): y abscissae
data ((np.array[N,M])): data corresponding the x,y
xlabel (string): x abscissa label
ylabel (string): y abscissa label
title (string): dataset title

Raises:

No exception is raised.

pyradi.ryfiles.**downloadFileUrl**(*url*, *saveFilename=None*)

Download a file, given a URL.

The URL is used to download a file, to the saveFilename specified. If no saveFilename is given, the basename of the URL is used. Before doownloading, first test to see if the file already exists.

Args:

url (string): the url to be accessed.
saveFilename (string): path to where the file must be saved (optional).

Returns:

(string): Filename saved, or None if failed.

Raises:

Exceptions are handled internally and signaled by return value.

pyradi.ryfiles.**unzipGZipfile**(*zipfilename*, *saveFilename=None*)

Unzip a file that was compressed using the gzip format.

The zipfilename is used to open a file, to the saveFilename specified. If no saveFilename is given, the basename of the zipfilename is used, but with the file extension removed.

Args:

zipfilename (string): the zipfilename to be decompressed.
saveFilename (string): to where the file must be saved (optional).

Returns:

(string): Filename saved, or None if failed.

Raises:

Exceptions are handled internally and signaled by return value.

pyradi.ryfiles.**untarTarfile**(*tarfilename*, *saveDirname=None*)
Untar a tar archive, and save all files to the specified directory.

The tarfilename is used to open a file, extraxting to the saveDirname specified. If no saveDirname is given, the local directory '.' is used.

Args:

tarfilename (string): the name of the tar archive.

saveDirname (string): to where the files must be extracted

Returns:

([string]): list of filenames saved, or None if failed.

Raises:

Exceptions are handled internally and signaled by return value.

pyradi.ryfiles.**downloadUntar**(*tgzFilename*, *url*, *destinationDir=None*, *tarFilename=None*)
Download and untar a compressed tar archive, and save all files to the specified directory.

The tarfilename is used to open the tar file, extracting to the destinationDir specified. If no destinationDir is given, the local directory '.' is used. Before downloading, a check is done to determine if the file was already downloaded and exists in the local file system.

Args:

tgzFilename (string): the name of the tar archive file

url (string): url where to look for the file (not including the filename)

destinationDir (string): to where the files must be extracted (optional)

tarFilename (string): downloaded tar filename (optional)

Returns:

([string]): list of filenames saved, or None if failed.

Raises:

Exceptions are handled internally and signaled by return value.

pyradi.ryfiles.**open_HDF**(*filename*)
Open and return an HDF5 file with the given filename.

See https://github.com/NelisW/pyradi/blob/master/pyradi/hdf5-as-data-format.md for more information on using HDF5 as a data structure.

Args:

filename (string): name of the file to be opened

Returns:

HDF5 file.

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryfiles.**erase_create_HDF**(*filename*)
Create and return a new HDS5 file with the given filename, erase the file if existing.

See https://github.com/NelisW/pyradi/blob/master/pyradi/hdf5-as-data-format.md for more information on using HDF5 as a data structure.

Args:

filename (string): name of the file to be created

Returns:

HDF5 file.

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryfiles.**print_HDF5_text**(*vartext*)
Prints text in visiting algorithm in HDF5 file.

See https://github.com/NelisW/pyradi/blob/master/pyradi/hdf5-as-data-format.md for more information on using HDF5 as a data structure.

Args:

vartext (string): string to be printed

Returns:

HDF5 file.

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryfiles.**print_HDF5_dataset_value**(*var*, *obj*)
Prints a data set in visiting algorithm in HDF5 file.

See https://github.com/NelisW/pyradi/blob/master/pyradi/hdf5-as-data-format.md for more information on using HDF5 as a data structure.

Args:

var (string): path to a dataset
obj (h5py dataset): dataset to be printed

Returns:

HDF5 file.

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryfiles.**get_HDF_branches**(*hdf5File*)
Print list of all the branches in the file

See https://github.com/NelisW/pyradi/blob/master/pyradi/hdf5-as-data-format.md for more information on using HDF5 as a data structure.

Args:

hdf5File (H5py file): the file to be opened

Returns:

HDF5 file.

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryfiles.**plotHDF5Bitmaps**(*hfd5f*, *prefix*, *format='png'*, *lstimgs=None*)

Plot arrays in the HFD5 as scaled bitmap images.

See https://github.com/NelisW/pyradi/blob/master/pyradi/hdf5-as-data-format.md for more information on using HDF5 as a data structure.

Retain zero in the array as black in the image, only scale the max value to 255

Args:

hfd5f (H5py file): the file to be opened
prefix (string): prefix to be prepended to filename
format (string): type of file to be created png/jpeg
lstimgs ([string]): list of paths to image in the HFD5 file

Returns:

Nothing.

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryfiles.**plotHDF5Images**(*hfd5f*, *prefix*, *colormap=<matplotlib.colors.LinearSegmentedColormap object at 0x00000000063331D0>*, *cbarshow=True*, *lstimgs=None*, *logscale=False*)

Plot images contained in hfd5f with colour map to show magnitude.

See https://github.com/NelisW/pyradi/blob/master/pyradi/hdf5-as-data-format.md for more information on using HDF5 as a data structure.

http://wiki.scipy.org/Cookbook/Matplotlib/Show_colormaps

Args:

hfd5f (H5py file): the file to be opened
prefix (string): prefix to be prepended to filename
colormap (Matplotlib colour map): colour map to be used in plot
cbarshow (boolean): indicate if colour bar must be shown
lstimgs ([string]): list of paths to image in the HFD5 file
logscale (boolean): True if display must be on log scale

Returns:

Nothing.

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryfiles.**plotHDF5Histograms**(*hfd5f*, *prefix*, *format='png'*, *lstimgs=None*, *bins=50*)

Plot histograms of images contained in hfd5f

See https://github.com/NelisW/pyradi/blob/master/pyradi/hdf5-as-data-format.md for more information on using HDF5 as a data structure.

Retain zero in the array as black in the image, only scale the max value to 255

Args:

hfd5f (H5py file): the file to be opened
prefix (string): prefix to be prepended to filename
format (string): type of file to be created png/jpeg
lstimgs ([string]): list of paths to image in the HFD5 file

bins ([int]): Number of bins to be used in histogram

Returns:

Nothing.

Raises:

No exception is raised.

Author: CJ Willers

CHAPTER

FOUR

PLOTTING UTILITY

4.1 Overview

This module provides functions for plotting cartesian and polar plots. This class provides a basic plotting capability, with a minimum number of lines. These are all wrapper functions, based on existing functions in other Python classes. Provision is made for combinations of linear and log scales, as well as polar plots for two-dimensional graphs. The Plotter class can save files to disk in a number of formats.

For more examples of use see: https://github.com/NelisW/ComputationalRadiometry

See the __main__ function for examples of use.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

4.2 Module classes

class `pyradi.ryplot.`**`Plotter`**(*fignumber=0*, *subpltnrow=1*, *subpltncol=1*, *figuretitle=None*, *figsize=(9, 9)*, *titlefontsize=14*)

Encapsulates a plotting environment, optimized for compact code.

This class provides a wrapper around Matplotlib to provide a plotting environment specialised towards typical pyradi visualisation. These functions were developed to provide sophisticated plots by entering the various plot options on a few lines, instead of typing many commands.

Provision is made for plots containing subplots (i.e., multiple plots on the same figure), linear scale and log scale plots, images, and cartesian, 3-D and polar plots.

`buildPlotCol`(*plotCol=None*, *n=None*)

Set a sequence of default colour styles of appropriate length.

The constructor provides a sequence with length 14 pre-defined plot styles. The user can define a new sequence if required. This function modulus-folds either sequence, in case longer sequences are required.

Colours can be one of the basic colours: ['b', 'g', 'r', 'c', 'm', 'y', 'k'] or it can be a gray shade float value between 0 and 1, such as '0.75', or it can be in hex format '#eeefff' or it can be one of the legal html colours. See http://html-color-codes.info/ and http://www.computerhope.com/htmcolor.htm.

Args:

plotCol ([strings]): User-supplied list
of plotting styles(can be empty []).
n (int): Length of required sequence.

Returns:

A list with sequence of plot styles, of required length.

Raises:

No exception is raised.

emptyPlot(*plotnum*, *projection='rectilinear'*)

Returns a handler to an empty plot.

This function does not do any plotting, the use must add plots using the standard MatPlotLib means.

Args:

plotnum (int): subplot number, 1-based index
rectilinear (str): type of axes projection, from ['aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear.].

Returns:

the axis object for the plot

Raises:

No exception is raised.

getPlot()

Returns a handle to the current figure

Args:

None

Returns:

A handle to the current figure.

Raises:

No exception is raised.

getSubPlot(*subplotNum=1*)

Returns a handle to the subplot, as requested per subplot number. Subplot numbers range from 1 upwards.

Args:

subplotNumer (int) : number of the subplot

Returns:

A handle to the requested subplot or None if not found.

Raises:

No exception is raised.

labelSubplot(*spax*, *ptitle=None*, *xlabel=None*, *ylabel=None*, *zlabel=None*, *titlefsize=10*, *labelfsize=10*)

Set the sub-figure title and axes labels (cartesian plots only).

Args:

spax (handle): subplot axis handle where labels must be drawn
ptitle (string): plot title (optional)
xlabel (string): x-axis label (optional)
ylabel (string): y-axis label (optional)
zlabel (string): z axis label (optional)
titlefsize (float): title fontsize (optional)
labelfsize (float): x,y,z label fontsize (optional)

Returns:

None.

Raises:

No exception is raised.

logLog(*plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], linewidths=None, label=[], legendAlpha=0.0, legendLoc='best', pltaxis=None, maxNX=10, maxNY=10, linestyle=None, powerLimits=[-4, 2, -4, 2], titlefsize=12, xylabelfsize=12, xytickfsize=10, labelfsize=10, xScientific=False, yScientific=False, yInvert=False, xInvert=False, drawGrid=True, xIsDate=False, xTicks=None, xtickRotation=0, markers=[], markevery=None, zorders=None, clip_on=True*)

Plot data on logarithmic scales for abscissa and ordinates.

Given an existing figure, this function plots in a specified subplot position. The function arguments are described below in some detail. Note that the y-values or ordinates can be more than one column, each column representing a different line in the plot. This is convenient if large arrays of data must be plotted. If more than one column is present, the label argument can contain the legend labels for each of the columns/lines. The pltaxis argument defines the min/max scale values for the x and y axes.

Args:

plotnum (int): subplot number, 1-based index

x (np.array[N,] or [N,1]): abscissa

y (np.array[N,] or [N,M]): ordinates - could be M columns

ptitle (string): plot title (optional)

xlabel (string): x-axis label (optional)

ylabel (string): y-axis label (optional)

plotCol ([strings]): plot colour and line style, list with M entries, use default if [] (optional)

linewidths ([float]): plot line width in points, list with M entries, use default if None (optional)

label ([strings]): legend label for ordinate, list with M entries (optional)

legendAlpha (float): transparency for legend box (optional)

legendLoc (string): location for legend box (optional)

pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. Let Matplotlib decide if None. (optional)

pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. Let Matplotlib decide if None. (optional)

maxNX (int): draw maxNX+1 tick labels on x axis (optional)

maxNY (int): draw maxNY+1 tick labels on y axis (optional)

linestyle (string): linestyle for this plot (optional)

powerLimits[float]: scientific tick label power limits [x-low, x-high, y-low, y-high] (optional) (optional)

titlefsize (int): title font size, default 12pt (optional)

xylabelfsize (int): x-axis, y-axis label font size, default 12pt (optional)

xytickfsize (int): x-axis, y-axis tick font size, default 10pt (optional)

labelfsize (int): label/legend font size, default 10pt (optional)

xScientific (bool): use scientific notation on x axis (optional)

yScientific (bool): use scientific notation on y axis (optional)

drawGrid (bool): draw the grid on the plot (optional)

yInvert (bool): invert the y-axis (optional)

xInvert (bool): invert the x-axis (optional)

xIsDate (bool): convert the datetime x-values to dates (optional)

xTicks ({tick:label}): dict of x-axis tick locations and associated labels (optional)

xtickRotation (float) x-axis tick label rotation angle (optional)

markers ([string]) markers to be used for plotting data points (optional)

markevery (int | (startind, stride)) subsample when using markers (optional)
zorders ([int]) list of zorder for drawing sequence, highest is last (optional)
clip_on (bool) clips objects to drawing axes (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

mesh3D (*plotnum, xvals, yvals, zvals, ptitle=None, xlabel=None, ylabel=None, zlabel=None, rstride=1, cstride=1, linewidth=0, plotCol=None, edgeCol=None, pltaxis=None, maxNX=0, maxNY=0, maxNZ=0, xScientific=False, yScientific=False, zScientific=False, powerLimits=[-4, 2, -4, 2, -2, 2], titlefsize=12, xylabelfsize=12, xytickfsize=10, wireframe=False, surface=True, cmap=<matplotlib.colors.LinearSegmentedColormap object at 0x00000000063C87B8>, cbarshow=False, cbarorientation='vertical', cbarcustomticks=[], cbarfontsize=12, drawGrid=True, xInvert=False, yInvert=False, zInvert=False, logScale=False, alpha=1, alphawire=1, azim=45, elev=30, distance=10, zorders=None, clip_on=True*)

XY colour mesh plot for (xvals, yvals, zvals) input sets.

Given an existing figure, this function plots in a specified subplot position. Only one mesh is drawn at a time. Future meshes in the same subplot will cover any previous meshes.

The mesh grid is defined in (x,y), while the height of the mesh is the z value.

The data set must have three two dimensional arrays, each for x, y, and z. The data in x, y, and z arrays must have matching data points. The x and y arrays each define the grid in terms of x and y values, i.e., the x array contains the x values for the data set, while the y array contains the y values. The z array contains the z values for the corresponding x and y values in the mesh.

Use wireframe=True to obtain a wireframe plot.

Use surface=True to obtain a surface plot with fill colours.

Z-values can be plotted on a log scale, in which case the colourbar is adjusted to show true values, but on the nonlinear scale.

The xvals and yvals vectors may have non-constant grid-intervals, i.e., they do not have to be on regular intervals, but z array must correspond to the (x,y) grid.

Args:

plotnum (int): subplot number, 1-based index
xvals (np.array[N,M]): array of x values, corresponding to (x,y) grid
yvals (np.array[N,M]): array of y values, corresponding to (x,y) grid
zvals (np.array[N,M]): array of z values, corresponding to (x,y) grid
ptitle (string): plot title (optional)
xlabel (string): x axis label (optional)
ylabel (string): y axis label (optional)
zlabel (string): z axis label (optional)
rstride (int): mesh line row (y axis) stride, every rstride value along y axis (optional)
cstride (int): mesh line column (x axis) stride, every cstride value along x axis (optional)
linewidth (float): mesh line width in points (optional)
plotCol ([strings]): fill colour, list with M=1 entries, use default if None (optional)
edgeCol ([strings]): mesh line colour , list with M=1 entries, use default if None (optional)
pltaxis ([xmin, xmax, ymin, ymax]): scale for x,y axes. z scale is not settable. Let Matplotlib decide if None (optional)
maxNX (int): draw maxNX+1 tick labels on x axis (optional)
maxNY (int): draw maxNY+1 tick labels on y axis (optional)

maxNZ (int): draw maxNY+1 tick labels on z axis (optional)
xScientific (bool): use scientific notation on x axis (optional)
yScientific (bool): use scientific notation on y axis (optional)
zScientific (bool): use scientific notation on z-axis (optional)
powerLimits[float]: scientific tick label power limits [x-neg, x-pos, y-neg, y-pos, z-neg, z-pos] (optional)
titlefsize (int): title font size, default 12pt (optional)
xylabelfsize (int): x-axis, y-axis, z-axis label font size, default 12pt (optional)
xytickfsize (int): x-axis, y-axis, z-axis tick font size, default 10pt (optional)
wireframe (bool): If True, do a wireframe plot, (optional)
surface (bool): If True, do a surface plot, (optional)
cmap (cm): color map for the mesh (optional)
cbarshow (bool): if true, the show a color bar (optional)
cbarorientation (string): 'vertical' (right) or 'horizontal' (below) (optional)
cbarcustomticks zip([z values/float],[tick labels/string]): define custom colourbar ticks locations for given z values(optional)
cbarfontsize (int): font size for color bar (optional)
drawGrid (bool): draw the grid on the plot (optional)
xInvert (bool): invert the x-axis. Flip the x-axis left-right (optional)
yInvert (bool): invert the y-axis. Flip the y-axis left-right (optional)
zInvert (bool): invert the z-axis. Flip the z-axis up-down (optional)
logScale (bool): do Z values on log scale, recompute colourbar vals (optional)
alpha (float): surface transparency (optional)
alphawire (float): mesh transparency (optional)
azim (float): graph view azimuth angle [degrees] (optional)
elev (float): graph view evelation angle [degrees] (optional)
distance (float): distance between viewer and plot (optional)
zorder ([int]) list of zorder for drawing sequence, highest is last (optional)
clip_on (bool) clips objects to drawing axes (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

meshContour(*plotnum, xvals, yvals, zvals, levels=10, ptitle=None, xlabel=None, ylabel=None, shading='flat', plotCol=[], pltaxis=None, maxNX=0, maxNY=0, xScientific=False, yScientific=False, powerLimits=[-4, 2, -4, 2], titlefsize=12, xylabelfsize=12, xytickfsize=10, meshCmap=<matplotlib.colors.LinearSegmentedColormap object at 0x00000000063C87B8>, cbarshow=False, cbarorientation='vertical', cbarcustomticks=[], cbarfontsize=12, drawGrid=False, yInvert=False, xInvert=False, contourFill=True, contourLine=True, logScale=False, negativeSolid=False, zeroContourLine=None, contLabel=False, contFmt='%.2f', contCol='k', contFonSz=8, contLinWid=0.5, zorders=None, clip_on=True*)

XY colour mesh countour plot for (xvals, yvals, zvals) input sets.

The data values must be given on a fixed mesh grid of three-dimensional (x,y,z) array input sets. The mesh grid is defined in (x,y), while the height of the mesh is the z value.

Given an existing figure, this function plots in a specified subplot position. Only one contour plot is drawn at a time. Future contours in the same subplot will cover any previous contours.

The data set must have three two dimensional arrays, each for x, y, and z. The data in x, y, and z arrays must have matching data points. The x and y arrays each define the grid in terms of x and y values, i.e., the x array contains the x values for the data set, while the y array contains the y values. The z

array contains the z values for the corresponding x and y values in the contour mesh.

Z-values can be plotted on a log scale, in which case the colourbar is adjusted to show true values, but on the nonlinear scale.

The current version only saves png files, since there appears to be a problem saving eps files.

The xvals and yvals vectors may have non-constant grid-intervals, i.e., they do not have to be on regular intervals.

Args:

plotnum (int): subplot number, 1-based index
xvals (np.array[N,M]): array of x values
yvals (np.array[N,M]): array of y values
zvals (np.array[N,M]): values on a (x,y) grid
levels (int or [float]): number of contour levels or a list of levels (optional)
ptitle (string): plot title (optional)
xlabel (string): x axis label (optional)
ylabel (string): y axis label (optional)
shading (string): not used currently (optional)
plotCol ([strings]): plot colour and line style, list with M entries, use default if [] (optional)
pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. Let Matplotlib decide if None. (optional)
maxNX (int): draw maxNX+1 tick labels on x axis (optional)
maxNY (int): draw maxNY+1 tick labels on y axis (optional)
xScientific (bool): use scientific notation on x axis (optional)
yScientific (bool): use scientific notation on y axis (optional)
powerLimits[float]: scientific tick label power limits [x-low, x-high, y-low, y-high] (optional)
titlefsize (int): title font size, default 12pt (optional)
xylabelfsize (int): x-axis, y-axis label font size, default 12pt (optional)
xytickfsize (int): x-axis, y-axis tick font size, default 10pt (optional)
meshCmap (cm): colour map for the mesh (optional)
cbarshow (bool): if true, the show a colour bar (optional)
cbarorientation (string): 'vertical' (right) or 'horizontal' (below) (optional)
cbarcustomticks zip([z values/float],[tick labels/string])' define custom colourbar ticks locations for given z values(optional)
cbarfontsize (int): font size for colour bar (optional)
drawGrid (bool): draw the grid on the plot (optional)
yInvert (bool): invert the y-axis. Flip the y-axis up-down (optional)
xInvert (bool): invert the x-axis. Flip the x-axis left-right (optional)
contourFill (bool): fill contours with colour (optional)
contourLine (bool): draw a series of contour lines (optional)
logScale (bool): do Z values on log scale, recompute colourbar values (optional)
negativeSolid (bool): draw negative contours in solid lines, dashed otherwise (optional)
zeroContourLine (double): draw a single contour at given value (optional)
contLabel (bool): label the contours with values (optional)
contFmt (string): contour label c-printf format (optional)
contCol (string): contour label colour, e.g., 'k' (optional)
contFonSz (float): contour label fontsize (optional)
contLinWid (float): contour line width in points (optional)
zorder ([int]) list of zorder for drawing sequence, highest is last (optional)
clip_on (bool) clips objects to drawing axes (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

myPlot (*plotcommand, plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], linewidths=None, label=[], legendAlpha=0.0, legendLoc='best', pltaxis=None, maxNX=0, maxNY=0, linestyle=None, powerLimits=[-4, 2, -4, 2], titlefsize=12, xylabelfsize=12, xytickfsize=10, labelfsize=10, drawGrid=True, xScientific=False, yScientific=False, yInvert=False, xInvert=False, xIsDate=False, xTicks=None, xtickRotation=0, markers=[], markevery=None, zorders=None, clip_on=True*)

Low level helper function to create a subplot and plot the data as required.

This function does the actual plotting, labelling etc. It uses the plotting function provided by its user functions.

lineStyles = { '': '_draw_nothing', ' ': '_draw_nothing', 'None': '_draw_nothing', '–': '_draw_dashed', '-.': '_draw_dash_dot', '-': '_draw_solid', ':': '_draw_dotted'}

Args:

plotcommand: name of a MatplotLib plotting function
plotnum (int): subplot number, 1-based index
ptitle (string): plot title
xlabel (string): x axis label
ylabel (string): y axis label
x (np.array[N,] or [N,1]): abscissa
y (np.array[N,] or [N,M]): ordinates - could be M columns
plotCol ([strings]): plot colour and line style, list with M entries, use default if []
linewidths ([float]): plot line width in points, list with M entries, use default if None (optional)
label ([strings]): legend label for ordinate, list with M entries
legendAlpha (float): transparency for legend box
legendLoc (string): location for legend box (optional)
pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. Let Matplotlib decide if None.
maxNX (int): draw maxNX+1 tick labels on x axis
maxNY (int): draw maxNY+1 tick labels on y axis
linestyle (string): linestyle for this plot (optional)
powerLimits[float]: scientific tick label power limits [x-low, x-high, y-low, y-high] (optional)
titlefsize (int): title font size, default 12pt (optional)
xylabelfsize (int): x-axis, y-axis label font size, default 12pt (optional)
xytickfsize (int): x-axis, y-axis tick font size, default 10pt (optional)
labelfsize (int): label/legend font size, default 10pt (optional)
drawGrid (bool): draw the grid on the plot (optional)
xScientific (bool): use scientific notation on x axis (optional)
yScientific (bool): use scientific notation on y axis (optional)
yInvert (bool): invert the y-axis (optional)
xInvert (bool): invert the x-axis (optional)
xIsDate (bool): convert the datetime x-values to dates (optional)
xTicks ({tick:label}): dict of x-axis tick locations and associated labels (optional)
xtickRotation (float) x-axis tick label rotation angle (optional)
markers ([string]) markers to be used for plotting data points (optional)
markevery (int | (startind, stride)) subsample when using markers (optional)
zorders ([int]) list of zorder for drawing sequence, highest is last (optional)

clip_on (bool) clips objects to drawing axes (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

nextPlotCol()

Returns the next entry in a sequence of default plot line colour styles in circular list. One day I want to do this with a generator....

Args:

None

Returns:

The next plot colour in the sequence.

Raises:

No exception is raised.

plot(*plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], linewidths=None, label=[], legendAlpha=0.0, legendLoc='best', pltaxis=None, maxNX=10, maxNY=10, linestyle=None, powerLimits=[-4, 2, -4, 2], titlefsize=12, xylabelfsize=12, xytickfsize=10, labelfsize=10, xScientific=False, yScientific=False, yInvert=False, xInvert=False, drawGrid=True, xIsDate=False, xTicks=None, xtickRotation=0, markers=[], markevery=None, zorders=None, clip_on=True*)

Cartesian plot on linear scales for abscissa and ordinates.

Given an existing figure, this function plots in a specified subplot position. The function arguments are described below in some detail. Note that the y-values or ordinates can be more than one column, each column representing a different line in the plot. This is convenient if large arrays of data must be plotted. If more than one column is present, the label argument can contain the legend labels for each of the columns/lines. The pltaxis argument defines the min/max scale values for the x and y axes.

Args:

plotnum (int): subplot number, 1-based index

x (np.array[N,] or [N,1]): abscissa

y (np.array[N,] or [N,M]): ordinates - could be M columns

ptitle (string): plot title (optional)

xlabel (string): x-axis label (optional)

ylabel (string): y-axis label (optional)

plotCol ([strings]): plot colour and line style, list with M entries, use default if [] (optional)

linewidths ([float]): plot line width in points, list with M entries, use default if None (optional)

label ([strings]): legend label for ordinate, list with M entries (optional)

legendAlpha (float): transparency for legend box (optional)

legendLoc (string): location for legend box (optional)

pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. Let Matplotlib decide if None. (optional)

maxNX (int): draw maxNX+1 tick labels on x axis (optional)

maxNY (int): draw maxNY+1 tick labels on y axis (optional)

linestyle (string): linestyle for this plot (optional)

powerLimits[float]: scientific tick label power limits [x-low, x-high, y-low, y-high] (optional)

titlefsize (int): title font size, default 12pt (optional)

xylabelfsize (int): x-axis, y-axis label font size, default 12pt (optional)
xytickfsize (int): x-axis, y-axis tick font size, default 10pt (optional)
labelfsize (int): label/legend font size, default 10pt (optional)
xScientific (bool): use scientific notation on x axis (optional)
yScientific (bool): use scientific notation on y axis (optional)
drawGrid (bool): draw the grid on the plot (optional)
yInvert (bool): invert the y-axis (optional)
xInvert (bool): invert the x-axis (optional)
xIsDate (bool): convert the datetime x-values to dates (optional)
xTicks ({tick:label}): dict of x-axis tick locations and associated labels (optional)
xtickRotation (float) x-axis tick label rotation angle (optional)
markers ([string]) markers to be used for plotting data points (optional)
markevery (int | (startind, stride)) subsample when using markers (optional)
zorders ([int]) list of zorder for drawing sequence, highest is last (optional)
clip_on (bool) clips objects to drawing axes (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

plot3d(*plotnum*, *x*, *y*, *z*, *ptitle=None*, *xlabel=None*, *ylabel=None*, *zlabel=None*, *plotCol=[]*, *linewidths=None*, *pltaxis=None*, *label=None*, *legendAlpha=0.0*, *titlefsize=12*, *xylabelfsize=12*, *xInvert=False*, *yInvert=False*, *zInvert=False*, *scatter=False*, *markers=None*, *markevery=None*, *azim=45*, *elev=30*, *zorders=None*, *clip_on=True*, *edgeCol=None*)

3D plot on linear scales for x y z input sets.

Given an existing figure, this function plots in a specified subplot position. The function arguments are described below in some detail.

Note that multiple 3D data sets can be plotted simultaneously by adding additional columns to the input coordinates of the (x,y,z) arrays, each set of columns representing a different line in the plot. This is convenient if large arrays of data must be plotted. If more than one column is present, the label argument can contain the legend labels for each of the columns/lines.

Args:

plotnum (int): subplot number, 1-based index
x (np.array[N,] or [N,M]) x coordinates of each line.
y (np.array[N,] or [N,M]) y coordinates of each line.
z (np.array[N,] or [N,M]) z coordinates of each line.
ptitle (string): plot title (optional)
xlabel (string): x-axis label (optional)
ylabel (string): y-axis label (optional)
zlabel (string): z axis label (optional)
plotCol ([strings]): plot colour and line style, list with M entries, use default if None (optional)
linewidths ([float]): plot line width in points, list with M entries, use default if None (optional)
pltaxis ([xmin, xmax, ymin, ymax, zmin, zmax]) scale for x,y,z axes. Let Matplotlib decide if None. (optional)
label ([strings]): legend label for ordinate, list with M entries (optional)
legendAlpha (float): transparency for legend box (optional)
titlefsize (int): title font size, default 12pt (optional)
xylabelfsize (int): x, y, z label font size, default 12pt (optional)

xInvert (bool): invert the x-axis (optional)
yInvert (bool): invert the y-axis (optional)
zInvert (bool): invert the z-axis (optional)
scatter (bool): draw only the points, no lines (optional)
markers ([string]): markers to be used for plotting data points (optional)
markevery (int | (startind, stride)): subsample when using markers (optional)
azim (float): graph view azimuth angle [degrees] (optional)
elev (float): graph view evelation angle [degrees] (optional)
zorder ([int]): list of zorder for drawing sequence, highest is last (optional)
clip_on (bool): clips objects to drawing axes (optional)
edgeCol ([int]): list of colour specs, value at [0] used for edge colour (optional).

Returns:

the axis object for the plot

Raises:

No exception is raised.

plotArray(*plotnum*, *inarray*, *slicedim=0*, *labels=None*, *maxNX=0*, *maxNY=0*, *titlefsize=8*, *xylabelfsize=8*, *xytickfsize=8*, *selectCols=None*, *sepSpace=0.2*, *allPlotCol='r'*)

Creates a plot from an input array.

Given an input array with m x n dimensions, this function creates a subplot for vectors [1-n]. Vector 0 serves as the x-axis for each subplot. The slice dimension can be in columns (0) or rows (1).

Args:

plotnum (int): The subplot number, 1-based index, according to Matplotlib conventions. This value must always be given, even if only a single 1,1 subplot is used.
inarray (np.array): data series to be plotted. Data direction can be cols or rows. The abscissa (x axis) values must be the first col/row, with ordinates in following cols/rows.
slicedim (int): slice along columns (0) or rows (1) (optional).
labels (list): a list of strings as labels for each subplot. x=labels[0], y=labels[1:] (optional).
maxNX (int): draw maxNX+1 tick labels on x axis (optional).
maxNY (int): draw maxNY+1 tick labels on y axis (optional).
titlefsize (int): title font size, default 12pt (optional).
xylabelfsize (int): x-axis, y-axis label font size, default 12pt (optional).
xytickfsize (int): x-axis, y-axis tick font size, default 10pt (optional).
selectCols ([int]): select columns for plot. Col 0 corresponds to col 1 in input data (because col 0 is abscissa),plot all if not given (optional).
sepSpace (float): vertical spacing between sub-plots in inches (optional).
allPlotCol (str): make all plot lines this colour (optional).

Returns:

Nothing

Raises:

No exception is raised.

polar(*plotnum, theta, r, ptitle=None, plotCol=None, label=[], labelLocation=[-0.1, 0.1], highlightNegative=True, highlightCol='#ffff00', highlightWidth=4, legendAlpha=0.0, rscale=None, rgrid=[0, 5], thetagrid=[30], direction='counterclockwise', zerooffset=0, titlefsize=12, drawGrid=True, zorders=None, clip_on=True, markers=[], markevery=None*)

Create a subplot and plot the data in polar coordinates (linear radial orginates only).

Given an existing figure, this function plots in a specified subplot position. The function arguments are described below in some detail. Note that the radial values or ordinates can be more than one column,

each column representing a different line in the plot. This is convenient if large arrays of data must be plotted. If more than one column is present, the label argument can contain the legend labels for each of the columns/lines. The scale for the radial ordinates can be set with rscale. The number of radial grid circles can be set with rgrid - this provides a somewhat better control over the built-in radial grid in matplotlib. thetagrids defines the angular grid interval. The angular rotation direction can be set to be clockwise or counterclockwise. Likewise, the rotation offset where the plot zero angle must be, is set with *zerooffset*.

For some obscure reason Matplitlib version 1.13 does not plot negative values on the polar plot. We therefore force the plot by making the values positive and then highlight it as negative.

Args:

plotnum (int): subplot number, 1-based index

theta (np.array[N,] or [N,1]): angular abscissa in radians

r (np.array[N,] or [N,M]): radial ordinates - could be M columns

ptitle (string): plot title (optional)

plotCol ([strings]): plot colour and line style, list with M entries, use default if None (optional)

label ([strings]): legend label, list with M entries (optional)

labelLocation ([x,y]): where the legend should located (optional)

highlightNegative (bool): indicate if negative data must be highlighted (optional)

highlightCol (string): negative highlight colour string (optional)

highlightWidth (int): negative highlight line width(optional)

legendAlpha (float): transparency for legend box (optional)

rscale ([rmin, rmax]): radial plotting limits. use default setting if None. If rmin is negative the zero is a circle and rmin is at the centre of the graph (optional)

rgrid ([rinc, numinc]): radial grid, use default is [0,5]. If rgrid is None don't show. If rinc=0 then numinc is number of intervals. If rinc is not zero then rinc is the increment and numinc is ignored (optional)

thetagrids (float): theta grid interval [degrees], if None don't show (optional)

direction (string): direction in increasing angle, 'counterclockwise' or 'clockwise' (optional)

zerooffset (float): rotation offset where zero should be [rad]. Positive zero-offset rotation is counterclockwise from 3'o'clock (optional)

titlefsize (int): title font size, default 12pt (optional)

drawGrid (bool): draw a grid on the graph (optional)

zorder ([int]) list of zorder for drawing sequence, highest is last (optional)

clip_on (bool) clips objects to drawing axes (optional)

markers ([string]) markers to be used for plotting data points (optional)

markevery (int | (startind, stride)) subsample when using markers (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

polar3d(*plotnum*, *theta*, *radial*, *zvals*, *ptitle=None*, *xlabel=None*, *ylabel=None*, *zlabel=None*, *zscale=None*, *titlefsize=12*, *xylabelfsize=12*, *thetaStride=1*, *radialstride=1*, *meshCmap=<matplotlib.colors.LinearSegmentedColormap object at 0x00000000063C87B8>*, *linewidth=0.1*, *azim=45*, *elev=30*, *zorders=None*, *clip_on=True*, *facecolors=None*, *alpha=1*, *edgeCol=None*)

3D polar surface/mesh plot for (r, theta, zvals) input sets.

Given an existing figure, this function plots in a specified subplot position.

Only one mesh is drawn at a time. Future meshes in the same subplot will cover any previous meshes.

The data in zvals must be on a grid where the theta vector correspond to the number of rows in zvals and the radial vector corresponds to the number of columns in zvals.

The r and p vectors may have non-constant grid-intervals, i.e., they do not have to be on regular intervals.

Args:

plotnum (int): subplot number, 1-based index
theta (np.array[N,M]): array of angular values [0..2pi] corresponding to (theta,rho) grid.
radial (np.array[N,M]): array of radial values corresponding to (theta,rho) grid.
zvals (np.array[N,M]): array of z values corresponding to (theta,rho) grid.
ptitle (string): plot title (optional)
xlabel (string): x-axis label (optional)
ylabel (string): y-axis label (optional)
zlabel (string): z-axis label (optional)
zscale ([float]): z axis [min, max] in the plot.
titlefsize (int): title font size, default 12pt (optional)
xylabelfsize (int): x, y, z label font size, default 12pt (optional)
thetaStride (int): theta stride in input data (optional)
radialstride (int): radial stride in input data (optional)
meshCmap (cm): color map for the mesh (optional)
linewidth (float): width of the mesh lines
azim (float): graph view azimuth angle [degrees] (optional)
elev (float): graph view evelation angle [degrees] (optional)
zorder ([int]) list of zorder for drawing sequence, highest is last (optional)
clip_on (bool) clips objects to drawing axes (optional)
facecolors ((np.array[N,M]): array of z value facecolours, corresponding to (theta,rho) grid.
alpha (float): facecolour surface transparency (optional)
edgeCol ([int]): list of colour specs, value at [0] used for edge colour (optional).

Returns:

the axis object for the plot

Raises:

No exception is raised.

polarMesh(*plotnum, theta, radial, zvals, ptitle=None, shading='flat', radscale=None, titlefsize=12, meshCmap=<matplotlib.colors.LinearSegmentedColormap object at 0x00000000063C87B8>, cbarshow=False, cbarorientation='vertical', cbarcustomticks=[], cbarfontsize=12, rgrid=[0, 5], thetagrid=[30], drawGrid=False, thetagridfontsize=12, radialgridfontsize=12, direction='counterclockwise', zerooffset=0, logScale=False, plotCol=[], levels=10, contourFill=True, contourLine=True, zeroContourLine=None, negativeSolid=False, contLabel=False, contFmt='%.2f', contCol='k', contFonSz=8, contLinWid=0.5, zorders=None, clip_on=True*)

Polar colour contour and filled contour plot for (theta, r, zvals) input sets.

The data values must be given on a fixed mesh grid of three-dimensional (theta,rho,z) array input sets (theta is angle, and rho is radial distance). The mesh grid is defined in (theta,rho), while the height of the mesh is the z value. The (theta,rho) arrays may have non-constant grid-intervals, i.e., they do not have to be on regular intervals.

Given an existing figure, this function plots in a specified subplot position. Only one contour plot is drawn at a time. Future contours in the same subplot will cover any previous contours.

The data set must have three two dimensional arrays, each for theta, rho, and z. The data in theta, rho, and z arrays must have matching data points. The theta and rho arrays each define the grid in terms of theta and rho values, i.e., the theta array contains the angular values for the data set, while the rho

array contains the radial values. The z array contains the z values for the corresponding theta and rho values in the contour mesh.

Z-values can be plotted on a log scale, in which case the colourbar is adjusted to show true values, but on the nonlinear scale.

The current version only saves png files, since there appears to be a problem saving eps files.

Args:

plotnum (int): subplot number, 1-based index
theta (np.array[N,M]) array of angular values [0..2pi] corresponding to (theta,rho) grid.
radial (np.array[N,M]) array of radial values corresponding to (theta,rho) grid.
zvals (np.array[N,M]) array of z values corresponding to (theta,rho) grid.
ptitle (string): plot title (optional)
shading (string): 'flat' | 'gouraud' (optional)
radscale ([float]): inner and outer radial scale max in the plot.
titlefsize (int): title font size, default 12pt (optional)
meshCmap (cm): color map for the mesh (optional)
cbarshow (bool): if true, the show a color bar
cbarorientation (string): 'vertical' (right) or 'horizontal' (below)
cbarcustomticks zip([tick locations/float],[tick labels/string]): locations in image grey levels
cbarfontsize (int): font size for color bar
rgrid ([float]): radial grid - None, [number], [inc,max]
thetagrid ([float]): angular grid - None, [inc]
drawGrid (bool): draw the grid on the plot (optional)
thetagridfontsize (float): font size for the angular grid
radialgridfontsize (float): font size for the radial grid
direction (string)= 'counterclockwise' or 'clockwise' (optional)
zerooffset (float) = rotation offset where zero should be [rad] (optional)
logScale (bool): do Z values on log scale, recompute colourbar vals
plotCol ([strings]): plot colour and line style, list with M entries, use default if []
levels (int or [float]): number of contour levels or a list of levels (optional)
contourFill (bool): fill contours with colour (optional)
contourLine (bool): draw a series of contour lines
zeroContourLine (double): draw a contour at the stated value (optional)
negativeSolid (bool): draw negative contours in solid lines, dashed otherwise (optional)
contLabel (bool): label the contours with values (optional)
contFmt (string): contour label c-printf format (optional)
contCol (string): contour label colour, e.g., 'k' (optional)
contFonSz (float): contour label fontsize (optional)
contLinWid (float): contour line width in points (optional)
zorder ([int]) list of zorder for drawing sequence, highest is last (optional)
clip_on (bool) clips objects to drawing axes (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

resetPlotCol()
Resets the plot colours to start at the beginning of the cycle.

Args:

None

Returns:

None.

Raises:

No exception is raised.

saveFig (*filename='mpl.png', dpi=300, bbox_inches='tight', pad_inches=0.1, useTrueType=True*)

Save the plot to a disk file, using filename, dpi specification and bounding box limits.

One of matplotlib's design choices is a bounding box strategy which may result in a bounding box that is smaller than the size of all the objects on the page. It took a while to figure this out, but the current default values for bbox_inches and pad_inches seem to create meaningful bounding boxes. These are however larger than the true bounding box. You still need a tool such as epstools or Adobe Acrobat to trim eps files to the true bounding box.

The type of file written is picked up in the filename. Most backends support png, pdf, ps, eps and svg.

Args:

filename (string): output filename to write plot, file ext

dpi (int): the resolution of the graph in dots per inch

bbox_inches: see matplotlib docs for more detail.

pad_inches: see matplotlib docs for more detail.

useTrueType: if True, truetype fonts are used in eps/pdf files, otherwise Type3

Returns:

Nothing. Saves a file to disk.

Raises:

No exception is raised.

semilogX (*plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], linewidths=None, label=[], legendAlpha=0.0, legendLoc='best', pltaxis=None, maxNX=10, maxNY=10, linestyle=None, powerLimits=[-4, 2, -4, 2], titlefsize=12, xylabelfsize=12, xytickfsize=10, labelfsize=10, xScientific=False, yScientific=False, yInvert=False, xInvert=False, drawGrid=True, xIsDate=False, xTicks=None, xtickRotation=0, markers=[], markevery=None, zorders=None, clip_on=True*)

Plot data on logarithmic scales for abscissa and linear scale for ordinates.

Given an existing figure, this function plots in a specified subplot position. The function arguments are described below in some detail. Note that the y-values or ordinates can be more than one column, each column representing a different line in the plot. This is convenient if large arrays of data must be plotted. If more than one column is present, the label argument can contain the legend labels for each of the columns/lines. The pltaxis argument defines the min/max scale values for the x and y axes.

Args:

plotnum (int): subplot number, 1-based index

x (np.array[N,] or [N,1]): abscissa

y (np.array[N,] or [N,M]): ordinates - could be M columns

ptitle (string): plot title (optional)

xlabel (string): x-axis label (optional)

ylabel (string): y-axis label (optional)

plotCol ([strings]): plot colour and line style, list with M entries, use default if [] (optional)

linewidths ([float]): plot line width in points, list with M entries, use default if None (optional)

label ([strings]): legend label for ordinate, list with M entries (optional)
legendAlpha (float): transparency for legend box (optional)
legendLoc (string): location for legend box (optional)
pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. Let Matplotlib decide if None. (optional)
maxNX (int): draw maxNX+1 tick labels on x axis (optional)
maxNY (int): draw maxNY+1 tick labels on y axis (optional)
linestyle (string): linestyle for this plot (optional)
powerLimits[float]: scientific tick label notation power limits [x-low, x-high, y-low, y-high] (optional) (optional)
titlefsize (int): title font size, default 12pt (optional)
xylabelfsize (int): x-axis, y-axis label font size, default 12pt (optional)
xytickfsize (int): x-axis, y-axis tick font size, default 10pt (optional)
labelfsize (int): label/legend font size, default 10pt (optional)
xScientific (bool): use scientific notation on x axis (optional)
yScientific (bool): use scientific notation on y axis (optional)
drawGrid (bool): draw the grid on the plot (optional)
yInvert (bool): invert the y-axis (optional)
xInvert (bool): invert the x-axis (optional)
xIsDate (bool): convert the datetime x-values to dates (optional)
xTicks ({tick:label}): dict of x-axis tick locations and associated labels (optional)
xtickRotation (float) x-axis tick label rotation angle (optional)
markers ([string]) markers to be used for plotting data points (optional)
markevery (int | (startind, stride)) subsample when using markers (optional)
zorders ([int]) list of zorder for drawing sequence, highest is last (optional)
clip_on (bool) clips objects to drawing axes (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

semilogY(*plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], linewidths=None, label=[], legendAlpha=0.0, legendLoc='best', pltaxis=None, maxNX=10, maxNY=10, linestyle=None, powerLimits=[-4, 2, -4, 2], titlefsize=12, xylabelfsize=12, xytickfsize=10, labelfsize=10, xScientific=False, yScientific=False, yInvert=False, xInvert=False, drawGrid=True, xIsDate=False, xTicks=None, xtickRotation=0, markers=[], markevery=None, zorders=None, clip_on=True*)

Plot data on linear scales for abscissa and logarithmic scale for ordinates.

Given an existing figure, this function plots in a specified subplot position. The function arguments are described below in some detail. Note that the y-values or ordinates can be more than one column, each column representing a different line in the plot. This is convenient if large arrays of data must be plotted. If more than one column is present, the label argument can contain the legend labels for each of the columns/lines. The pltaxis argument defines the min/max scale values for the x and y axes.

Args:

plotnum (int): subplot number, 1-based index
x (np.array[N,] or [N,1]): abscissa
y (np.array[N,] or [N,M]): ordinates - could be M columns
ptitle (string): plot title (optional)
xlabel (string): x-axis label (optional)
ylabel (string): y-axis label (optional)

plotCol ([strings]): plot colour and line style, list with M entries, use default if [] (optional)
linewidths ([float]): plot line width in points, list with M entries, use default if None (optional)
label ([strings]): legend label for ordinate, list withM entries (optional)
legendAlpha (float): transparency for legend box (optional)
legendLoc (string): location for legend box (optional)
pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. Let Matplotlib decide if None. (optional)
maxNX (int): draw maxNX+1 tick labels on x axis (optional)
maxNY (int): draw maxNY+1 tick labels on y axis (optional)
linestyle (string): linestyle for this plot (optional)
powerLimits[float]: scientific tick label power limits [x-low, x-high, y-low, y-high] (optional) (optional)
titlefsize (int): title font size, default 12pt (optional)
xylabelfsize (int): x-axis, y-axis label font size, default 12pt (optional)
xytickfsize (int): x-axis, y-axis tick font size, default 10pt (optional)
labelfsize (int): label/legend font size, default 10pt (optional)
xScientific (bool): use scientific notation on x axis (optional)
yScientific (bool): use scientific notation on y axis (optional)
drawGrid (bool): draw the grid on the plot (optional)
yInvert (bool): invert the y-axis (optional)
xInvert (bool): invert the x-axis (optional)
xIsDate (bool): convert the datetime x-values to dates (optional)
xTicks ({tick:label}): dict of x-axis tick locations and associated labels (optional)
xtickRotation (float) x-axis tick label rotation angle (optional)
markers ([string]) markers to be used for plotting data points (optional)
markevery (int | (startind, stride)) subsample when using markers (optional)
zorders ([int]) list of zorder for drawing sequence, highest is last (optional)
clip_on (bool) clips objects to drawing axes (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

showImage(*plotnum*, *img*, *ptitle=None*, *xlabel=None*, *ylabel=None*, *cmap=<matplotlib.colors.LinearSegmentedColormap object at 0x00000000063BF128>*, *titlefsize=12*, *cbarshow=False*, *cbarorientation='vertical'*, *cbarcustomticks=[]*, *cbarfontsize=12*, *labelfsize=10*, *xylabelfsize=12*)

Creates a subplot and show the image using the colormap provided.

Args:

plotnum (int): subplot number, 1-based index
img (np.ndarray): numpy 2d array containing the image
ptitle (string): plot title (optional)
xlabel (string): x axis label (optional)
ylabel (string): y axis label (optional)
cmap: matplotlib colormap, default gray (optional)
fsize (int): title font size, default 12pt (optional)
cbarshow (bool): if true, the show a colour bar (optional)
cbarorientation (string): 'vertical' (right) or 'horizontal' (below) (optional)

cbarcustomticks zip([tick locations/float],[tick labels/string]): locations in image grey levels (optional)

cbarfontsize (int): font size for colour bar (optional)

titlefsize (int): title font size, default 12pt (optional)

xylabelfsize (int): x-axis, y-axis label font size, default 12pt (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

stackplot (*plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], linewidths=None, label=[], legendAlpha=0.0, legendLoc='best', pltaxis=None, maxNX=10, maxNY=10, linestyle=None, powerLimits=[-4, 2, -4, 2], titlefsize=12, xylabelfsize=12, xytickfsize=10, labelfsize=10, xScientific=False, yScientific=False, yInvert=False, xInvert=False, drawGrid=True, xIsDate=False, xTicks=None, xtickRotation=0, markers=[], markevery=None, zorders=None, clip_on=True*)

Plot stacked data on linear scales for abscissa and ordinates.

Given an existing figure, this function plots in a specified subplot position. The function arguments are described below in some detail. Note that the y-values or ordinates can be more than one column, each column representing a different line in the plot. If more than one column is present, the label argument can contain the legend labels for each of the columns/lines. The pltaxis argument defines the min/max scale values for the x and y axes.

Args:

plotnum (int): subplot number, 1-based index

x (np.array[N,] or [N,1]): abscissa

y (np.array[N,] or [N,M]): ordinates - could be M columns

ptitle (string): plot title (optional)

xlabel (string): x-axis label (optional)

ylabel (string): y-axis label (optional)

plotCol ([strings]): plot colour and line style, list with M entries, use default if [] (optional)

linewidths ([float]): plot line width in points, list with M entries, use default if None (optional)

label ([strings]): legend label for ordinate, list withM entries (optional)

legendAlpha (float): transparency for legend box (optional)

legendLoc (string): location for legend box (optional)

pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. Let Matplotlib decide if None. (optional)

maxNX (int): draw maxNX+1 tick labels on x axis (optional)

maxNY (int): draw maxNY+1 tick labels on y axis (optional)

linestyle (string): linestyle for this plot (optional)

powerLimits[float]: scientific tick label power limits [x-low, x-high, y-low, y-high] (optional) (optional)

titlefsize (int): title font size, default 12pt (optional)

xylabelfsize (int): x-axis, y-axis label font size, default 12pt (optional)

xytickfsize (int): x-axis, y-axis tick font size, default 10pt (optional)

labelfsize (int): label/legend font size, default 10pt (optional)

xScientific (bool): use scientific notation on x axis (optional)

yScientific (bool): use scientific notation on y axis (optional)

drawGrid (bool): draw the grid on the plot (optional)

yInvert (bool): invert the y-axis (optional)

xInvert (bool): invert the x-axis (optional)
xIsDate (bool): convert the datetime x-values to dates (optional)
xTicks ({tick:label}): dict of x-axis tick locations and associated labels (optional)
xtickRotation (float) x-axis tick label rotation angle (optional)
markers ([string]) markers to be used for plotting data points (optional)
markevery (int | (startind, stride)) subsample when using markers (optional)
zorders ([int]) list of zorder for drawing sequence, highest is last (optional)
clip_on (bool) clips objects to drawing axes (optional)

Returns:

the axis object for the plot

Raises:

No exception is raised.

class pyradi.ryplot.**FilledMarker**(*markerfacecolor=None*, *markerfacecoloralt=None*, *markeredgecolor=None*, *marker=None*, *markersize=None*, *fillstyle=None*)

Filled marker user-settable values.

This class encapsulates a few variables describing a Filled marker. Default values are provided that can be overridden in user plots.

Values relevant to filled makers are as follows:

marker = ['o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd']
fillstyle = ['full', 'left', 'right', 'bottom', 'top', 'none']
colour names = http://www.w3schools.com/html/html_colornames.asp

class pyradi.ryplot.**Markers**(*markerfacecolor=None*, *markerfacecoloralt=None*, *markeredgecolor=None*, *marker=None*, *markersize=None*, *fillstyle=None*)

Collect marker location and types and mark subplot.

Build a list of markers at plot locations with the specified marker.

add(*x*, *y*, *markerfacecolor=None*, *markerfacecoloralt=None*, *markeredgecolor=None*, *marker=None*, *markersize=None*, *fillstyle=None*)

Add a marker to the list, overridding properties if necessary.

Specify location and any specific marker properties to be used. The location can be (xy,y) for cartesian plots or (theta,rad) for polars.

If no marker properties are specified, the current marker class properties will be used. If the current maker instance does not specify properties, the default marker properties will be used.

Args:

x (float): the x/theta location for the marker
y (float): the y/radial location for the marker
markerfacecolor (colour): main colour for marker (optional)
markerfacecoloralt (colour): alterive colour for marker (optional)
markeredgecolor (colour): edge colour for marker (optional)
marker (string): string to specify the marker (optional)
markersize (int)): size of the marker (optional)
fillstyle (string): string to define fill style (optional)

Returns:

Nothing. Creates the figure for subequent use.

Raises:

No exception is raised.

plot(*ax*)

Plot the current list of markers on the given axes.

All the markers currently stored in the class will be drawn.

Args:

ax (axes): an axes handle for the plot

Returns:

Nothing. Creates the figure for subequent use.

Raises:

No exception is raised.

class pyradi.ryplot.**ProcessImage**

This class provides a functions to assist in the optimal display of images.

compressEqualizeImage(*image*, *selectCompressSet=2*, *numCbarlevels=20*, *cbarformat='.3f'*)

Compress an image (and then inversely expand the color bar values), prior to histogram equalisation to ensure that the two keep in step, we store the compression function names as pairs, and invoke the compression function as follows: linear, log. sqrt. Note that the image is histogram equalised in all cases.

Args:

image (np.ndarray): the image to be processed

selectCompressSet (int): compression selection [0,1,2] (optional)

numCbarlevels (int): number of labels in the colourbar (optional)

cbarformat (string): colourbar label format, e.g., '10.3f', '.5e' (optional)

Returns:

imgHEQ (np.ndarray): the equalised image array

customticksz (zip(float, string)): colourbar levels and associated levels

Raises:

No exception is raised.

reprojectImageIntoPolar(*data*, *origin=None*, *framesFirst=True*, *cval=0.0*)

Reprojects a 3D numpy array into a polar coordinate system, relative to some origin.

This function reprojects an image from cartesian to polar coordinates. The origin of the new coordinate system defaults to the center of the image, unless the user supplies a new origin.

The data format can be data.shape = (rows, cols, frames) or data.shape = (frames, rows, cols), the format of which is indicated by the framesFirst parameter.

The reprojectImageIntoPolar function maps radial to cartesian coords. The radial image is however presented in a cartesian grid, the corners have no meaning. The radial coordinates are mapped to the radius, not the corners. This means that in order to map corners, the frequency is scaled with sqrt(2), The corners are filled with the value specified in cval.

Args:

data (np.array): 3-D array to which transformation must be applied.

origin ((x-orig, y-orig)): data-coordinates of where origin should be placed

framesFirst (bool): True if data.shape is (frames, rows, cols), False if data.shape is (rows, cols, frames)

cval (float): the fill value to be used in coords outside the mapped range(optional)

Returns:

output (float np.array): transformed images/array data in the same sequence as input sequence.

r_i (np.array[N,]): radial values for returned image.
theta_i (np.array[M,]): angular values for returned image.

Raises:

No exception is raised.

original code by Joe Kington https://stackoverflow.com/questions/3798333/image-information-along-a-polar-coordinate-system

4.3 Module functions

pyradi.ryplot.**savePlot**(**args*, ***kwds*)

Uses 'with' statement to create a plot and save to file on exit.

Use as follows:

```
x=np.linspace(-3,3,20)
with savePlot(1,saveName=['testwith.png','testwith.eps']) as p:
    p.plot(1,x,x*x)
```

where the savePlot parameters are exactly the same as `Plotter`, except that a new named parameter `saveName` is now present. If `saveName` is not `None`, the list of filenames is used to save files of the plot (any number of names/types)

Args:

fignumber (int): the plt figure number, must be supplied
subpltnrow (int): subplot number of rows
subpltncol (int): subplot number of columns
figuretitle (string): the overall heading for the figure
figsize ((w,h)): the figure size in inches
saveName str or [str]: string or list of save filenames

Returns:

The plotting object, used to populate the plot (see example)

Raises:

No exception is raised.

pyradi.ryplot.**cubehelixcmap**(*start=0.5*, *rot=-1.5*, *gamma=1.0*, *hue=1.2*, *reverse=False*, *nlev=256.0*)

A full implementation of Dave Green's "cubehelix" for Matplotlib. Based on the FORTRAN 77 code provided in D.A. Green, 2011, BASI, 39, 289.

http://adsabs.harvard.edu/abs/2011arXiv1108.5083G http://www.astron-soc.in/bulletin/11June/289392011.pdf

User can adjust all parameters of the cubehelix algorithm. This enables much greater flexibility in choosing color maps, while always ensuring the color map scales in intensity from black to white. A few simple examples:

Default color map settings produce the standard "cubehelix".

Create color map in only blues by setting rot=0 and start=0.

Create reverse (white to black) backwards through the rainbow once by setting rot=1 and reverse=True.

Args:

start : scalar, optional
Sets the starting position in the color space. 0=blue, 1=red,
2=green. Defaults to 0.5.

rot : scalar, optional
The number of rotations through the rainbow. Can be positive
or negative, indicating direction of rainbow. Negative values
correspond to Blue->Red direction. Defaults to -1.5
gamma : scalar, optional
The gamma correction for intensity. Defaults to 1.0
hue : scalar, optional
The hue intensity factor. Defaults to 1.2
reverse : boolean, optional
Set to True to reverse the color map. Will go from black to
white. Good for density plots where shade~density. Defaults to False
nevl : scalar, optional
Defines the number of discrete levels to render colors at.
Defaults to 256.

Returns:

matplotlib.colors.LinearSegmentedColormap object

Example: >>> import cubehelix >>> cx = cubehelix.cmap(start=0., rot=-0.5) >>> plot(x,cmap=cx)

Revisions 2014-04 (@jradavenport) Ported from IDL version

source https://github.com/jradavenport/cubehelix

CHAPTER

FIVE

SPHERICAL PLOTTING UTILITY

5.1 Overview

This module will be removed in future. There seems to be no movement in porting Mayavi to Python 3.

#.. automodule:: pyradi.ryplotspherical

#Module functions #————————

#.. autofunction:: pyradi.ryplotspherical.readOffFile

#.. autofunction:: pyradi.ryplotspherical.getRotateFromOffFile

#.. autofunction:: pyradi.ryplotspherical.getOrbitFromOffFile

#.. autofunction:: pyradi.ryplotspherical.writeOSSIMTrajOFFFile

#.. autofunction:: pyradi.ryplotspherical.writeOSSIMTrajElevAzim

#.. autofunction:: pyradi.ryplotspherical.getOrbitFromElevAzim

#.. autofunction:: pyradi.ryplotspherical.getRotateFromElevAzim

#.. autofunction:: pyradi.ryplotspherical.plotSpherical

#.. autofunction:: pyradi.ryplotspherical.plotOSSIMSpherical

#.. autofunction:: pyradi.ryplotspherical.sphericalPlotElevAzim

#.. autofunction:: pyradi.ryplotspherical.polarPlotElevAzim

#.. autofunction:: pyradi.ryplotspherical.globePlotElevAzim

#.. autofunction:: pyradi.ryplotspherical.plotVertexSphere

CHAPTER

SIX

UTILITY FUNCTIONS

6.1 Overview

This module provides various utility functions for radiometry calculations. Functions are provided for a maximally flat spectral filter, a simple photon detector spectral response, effective value calculation, conversion of spectral domain variables between [um], [cm^-1] and [Hz], conversion of spectral density quantities between [um], [cm^-1] and [Hz] and spectral convolution.

See the __main__ function for examples of use.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

6.2 Module classes

class pyradi.ryutils.**Spectral**(*ID*, *value*, *wl=None*, *wn=None*, *desc=None*)

Generic spectral can be used for any spectral vector

plot(*filename=None*, *heading=None*, *ytitle=''*)

Do a simple plot of spectral variable(s)

Args:

filename (str): filename for png graphic
heading (str): graph heading
ytitle (str): graph y-axis title

Returns:

Nothing, writes png file to disk

Raises:

No exception is raised.

vecalign(*other*)

returns two spectral values properly interpolated and aligned to same base

it is not intended that the function will be called directly by the user

Args:

other (Spectral): the other Spectral to be used in addition

Returns:

wl, wn, s, o

Raises:

No exception is raised.

class pyradi.ryutils.**Atmo**(*ID*, *distance=None*, *wl=None*, *wn=None*, *tran=None*, *atco=None*, *prad=None*, *desc=None*)

Atmospheric spectral such as transittance or attenuation coefficient

pathR(*distance*)

Calculates the path radiance at distance

Distance is in m

Args:

distance (scalar or np.array (M,)): distance in m if transmittance, or None if att coeff

Returns:

transmittance (np.array (N,M)): transmittance along N at distance along M

Raises:

No exception is raised.

tauR(*distance*)

Calculates the transmittance at distance

Distance is in m

Args:

distance (scalar or np.array (M,)): distance in m if transmittance, or None if att coeff

Returns:

transmittance (np.array (N,M)): transmittance along N at distance along M

Raises:

No exception is raised.

class pyradi.ryutils.**Sensor**(*ID*, *fno*, *detarea*, *inttime*, *tauOpt=1*, *quantEff=1*, *pfrac=1*, *desc=''*)

Sensor characteristics

QE()

Returns scaler or np.array for detector quantEff

Args:

None

Returns:

str

Raises:

No exception is raised.

tauOpt()

Returns scaler or np.array for optics transmittance

Args:

None

Returns:

str

Raises:

No exception is raised.

class pyradi.ryutils.**Target**(*ID*, *tmprt*, *emis*, *refl=1*, *cosTarg=1*, *taumed=1*, *scale=1*, *desc=''*)

Target / Source characteristics

emis()
Returns scaler or np.array for emissivity

Args:

None

Returns:

str

Raises:

No exception is raised.

radiance(*units='el'*)
Returns radiance spectral for target

The type of spectral is one of the following: type='el [W/(m^2.mum)] type='ql' [q/(s.m^2.mum)] type='en' [W/(m^2.cm^{-1})] type='qn' [q/(s.m^2.cm^{-1})]

Args:

None

Returns:

str

Raises:

No exception is raised.

refl()
Returns scaler or np.array for reflectance

Args:

None

Returns:

str

Raises:

No exception is raised.

taumed()
Returns scaler or np.array for atmospheric transmittance to illuminating source

Args:

None

Returns:

str

Raises:

No exception is raised.

6.3 Module functions

pyradi.ryutils.**sfilter**(*spectral*, *center*, *width*, *exponent=6*, *taupass=1.0*, *taustop=0.0*, *filtertype='bandpass'*)
Calculate a symmetrical filter response of shape exp(-x^n)

Given a number of parameters, calculates maximally flat, symmetrical transmittance. The function parameters controls the width, pass-band and stop-band transmittance and sharpness of cutoff. This function is

not meant to replace the use of properly measured filter responses, but rather serves as a starting point if no other information is available. This function does not calculate ripple in the pass-band or cut-off band.

Filter types supported include band pass, high (long) pass and low (short) pass filters. High pass filters have maximal transmittance for all spectral values higher than the central value. Low pass filters have maximal transmittance for all spectral values lower than the central value.

Args:

spectral (np.array[N,] or [N,1]): spectral vector in [um] or [cm-1].
center (float): central value for filter passband
width (float): proportional to width of filter passband
exponent (float): even integer, define the sharpness of cutoff.
If exponent=2 then gaussian
If exponent=infinity then square
taupass (float): the transmittance in the pass band (assumed constant)
taustop (float): peak transmittance in the stop band (assumed constant)
filtertype (string): filter type, one of 'bandpass', 'lowpass' or 'highpass'

Returns:

transmittance (np.array[N,] or [N,1]): transmittances at "spectral" intervals.

Raises:

No exception is raised.
If an invalid filter type is specified, return None.
If negative spectral is specified, return None.

pyradi.ryutils.**responsivity**(*wavelength*, *lwavepeak*, *cuton=1*, *cutoff=20*, *scaling=1.0*)
Calculate a photon detector wavelength spectral responsivity

Given a number of parameters, calculates a shape that is somewhat similar to a photon detector spectral response, on wavelength scale. The function parameters controls the cutoff wavelength and shape of the response. This function is not meant to replace the use of properly measured spectral responses, but rather serves as a starting point if no other information is available.

Args:

wavelength (np.array[N,] or [N,1]): vector in [um].
lwavepeak (float): approximate wavelength at peak response
cutoff (float): cutoff strength beyond peak, 5 < cutoff < 50
cuton (float): cuton sharpness below peak, 0.5 < cuton < 5
scaling (float): scaling factor

Returns:

responsivity (np.array[N,] or [N,1]): responsivity at wavelength intervals.

Raises:

No exception is raised.

pyradi.ryutils.**effectiveValue**(*spectraldomain*, *spectralToProcess*, *spectralBaseline*)
Normalise a spectral quantity to a scalar, using a weighted mapping by another spectral quantity.

Effectivevalue = integral(spectralToProcess * spectralBaseline) / integral(spectralBaseline)

The data in spectralToProcess and spectralBaseline must both be sampled at the same domain values as specified in spectraldomain.

The integral is calculated with numpy/scipy trapz trapezoidal integration function.

Args:

inspectraldomain (np.array[N,] or [N,1]): spectral domain in wavelength, frequency or wavenumber.

spectralToProcess (np.array[N,] or [N,1]): spectral quantity to be normalised
spectralBaseline (np.array[N,] or [N,1]): spectral serving as baseline for normalisation

Returns:

(float): effective value
Returns None if there is a problem

Raises:

No exception is raised.

pyradi.ryutils.**convertSpectralDomain**(*inspectraldomain*, *type=''*)
Convert spectral domains, i.e. between wavelength [um], wavenummber [cm^-1] and frequency [Hz]

In string variable type, the 'from' domain and 'to' domains are indicated each with a single letter: 'f' for temporal frequency, 'l' for wavelength and 'n' for wavenumber The 'from' domain is the first letter and the 'to' domain the second letter.

Note that the 'to' domain vector is a direct conversion of the 'from' domain to the 'to' domain (not interpolated or otherwise sampled.

Args:

inspectraldomain (np.array[N,] or [N,1]): spectral domain in wavelength, frequency or wavenumber.
wavelength vector in [um]
frequency vector in [Hz]
wavenumber vector in [cm^-1]
type (string): specify from and to domains:
'lf' convert from wavelength to per frequency
'ln' convert from wavelength to per wavenumber
'fl' convert from frequency to per wavelength
'fn' convert from frequency to per wavenumber
'nl' convert from wavenumber to per wavelength
'nf' convert from wavenumber to per frequency

Returns:

[N,1]: outspectraldomain
Returns zero length array if type is illegal, i.e. not one of the expected values

Raises:

No exception is raised.

pyradi.ryutils.**convertSpectralDensity**(*inspectraldomain*, *inspectralquantity*, *type=''*)
Convert spectral density quantities, i.e. between W/(m^2.um), W/(m^2.cm^-1) and W/(m^2.Hz).

In string variable type, the 'from' domain and 'to' domains are indicated each with a single letter: 'f' for temporal frequency, 'w' for wavelength and ''n' for wavenumber The 'from' domain is the first letter and the 'to' domain the second letter.

The return values from this function are always positive, i.e. not mathematically correct, but positive in the sense of radiance density.

The spectral density quantity input is given as a two vectors: the domain value vector and the density quantity vector. The output of the function is also two vectors, i.e. the 'to' domain value vector and the 'to' spectral density. Note that the 'to' domain vector is a direct conversion of the 'from' domain to the 'to' domain (not interpolated or otherwise sampled).

Args:

inspectraldomain (np.array[N,] or [N,1]): spectral domain in wavelength, frequency or wavenumber.
inspectralquantity (np.array[N,] or [N,1]): spectral density in same domain as domain vector above.
wavelength vector in [um]

frequency vector in [Hz]
wavenumber vector in [cm^-1]

type (string): specify from and to domains:
'lf' convert from per wavelength interval density to per frequency interval density
'ln' convert from per wavelength interval density to per wavenumber interval density
'fl' convert from per frequency interval density to per wavelength interval density
'fn' convert from per frequency interval density to per wavenumber interval density
'nl' convert from per wavenumber interval density to per wavelength interval density
'nf' convert from per wavenumber interval density to per frequency interval density

Returns:

([N,1],[N,1]): outspectraldomain and outspectralquantity
Returns zero length arrays is type is illegal, i.e. not one of the expected values

Raises:

No exception is raised.

pyradi.ryutils.**convolve**(*inspectral*, *samplingresolution*, *inwinwidth*, *outwinwidth*, *windowtype=<function bartlett at 0x00000000058C26D8>*)

Convolve (non-circular) a spectral variable with a window function, given the input resolution and input and output window widths.

This function is normally used on wavenumber-domain spectral data. The spectral data is assumed sampled at samplingresolution wavenumber intervals. The inwinwidth and outwinwidth window function widths are full width half-max (FWHM) for the window functions for the inspectral and returned spectral variables, respectively. The Bartlett function is used as default, but the user can use a different function. The Bartlett function is a triangular function reaching zero at the ends. Window functio width is correct for Bartlett and only approximate for other window functions.

Args:

inspectral (np.array[N,] or [N,1]): vector in [cm-1].
samplingresolution (float): wavenumber interval between inspectral samples
inwinwidth (float): FWHM window width of the input spectral vector
outwinwidth (float): FWHM window width of the output spectral vector
windowtype (function): name of a numpy/scipy function for the window function

Returns:

outspectral (np.array[N,]): input vector, filtered to new window width.
windowfn (np.array[N,]): The window function used.

Raises:

No exception is raised.

pyradi.ryutils.**savitzkyGolay1D**(*y*, *window_size*, *order*, *deriv=0*, *rate=1*)

Smooth (and optionally differentiate) data with a Savitzky-Golay filter.

Source: http://wiki.scipy.org/Cookbook/SavitzkyGolay

The Savitzky Golay filter is a particular type of low-pass filter, well adapted for data smoothing. For further information see: http://www.wire.tu-bs.de/OLDWEB/mameyer/cmr/savgol.pdf

The Savitzky-Golay filter removes high frequency noise from data. It has the advantage of preserving the original shape and features of the signal better than other types of filtering approaches, such as moving averages techniques.

The Savitzky-Golay is a type of low-pass filter, particularly suited for smoothing noisy data. The main idea behind this approach is to make for each point a least-square fit with a polynomial of high order over a odd-sized window centered at the point.

Examples: t = np.linspace(-4, 4, 500) y = np.exp(-t**2) + np.random.normal(0, 0.05, t.shape) ysg = savitzky_golay(y, window_size=31, order=4) import matplotlib.pyplot as plt plt.plot(t, y, label='Noisy signal') plt.plot(t, np.exp(-t**2), 'k', lw=1.5, label='Original signal') plt.plot(t, ysg, 'r', label='Filtered signal') plt.legend() plt.show()

References:

[1] A. Savitzky, M. J. E. Golay, Smoothing and Differentiation of Data by Simplified Least Squares Procedures. Analytical Chemistry, 1964, 36 (8), pp 1627-1639.

[2] Numerical Recipes 3rd Edition: The Art of Scientific Computing W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery Cambridge University Press ISBN-13: 9780521880688

Args:

y : array_like, shape (N,) the values of the time history of the signal.

window_size : int the length of the window. Must be an odd integer number.

order : int the order of the polynomial used in the filtering. Must be less then *window_size* - 1.

deriv: int the order of the derivative to compute (default = 0 means only smoothing)

Returns:

ys : ndarray, shape (N) the smoothed signal (or it's n-th derivative).

Raises:

Exception raised for window size errors.

pyradi.ryutils.**abshumidity**(*T*, *equationSelect=1*)

Atmopsheric absolute humidity [g/m3] for temperature in [K] between 248 K and 342 K.

This function provides two similar equations, but with different constants.

Args:

temperature (np.array[N,] or [N,1]): in [K].

equationSelect (int): select the equation to be used.

Returns:

absolute humidity (np.array[N,] or [N,1]): abs humidity in [g/m3]

Raises:

No exception is raised.

pyradi.ryutils.**rangeEquation**(*Intensity*, *Irradiance*, *rangeTab*, *tauTab*, *rangeGuess=1*, *n=2*)

Solve the range equation for arbitrary transmittance vs range.

This function solve for the range R in the range equation

$$E = \frac{I\tau_a(R)}{R^n}$$

where E is the threshold irradiance in [W/m2], and I is the intensity in [W/sr]. This range equation holds for the case where the target is smaller than the field of view.

The range R must be in [m], and $\tau_a(R)$ is calculated from a lookup table of atmospheric transmittance vs. range. The transmittance lookup table can be calculated from the simple Bouguer law, or it can have any arbitrary shape, provided it decreases with increasing range. The user supplies the lookup table in the form of an array of range values and an associated array of transmittance values. The range values need not be on constant linear range increment.

The parameter n

- $n = 2$ (default value) the general case of a radiating source smaller than the field of view.

- $n = 4$ the special case of a laser range finder illuminating a target smaller than the field of view, viewed against the sky. In this case there is an R^2 attenuation from the laser to the source and another R^2 attenuation from the source to the receiver, hence R^4 overall.

If the range solution is doubtful (e.g. not a trustworthy solution) the returned value is made negative.

Args:

Intensity (float or np.array[N,] or [N,1]): in [W/sr].

Irradiance (float or np.array[N,] or [N,1]): in [W/m2].

rangeTab (np.array[N,] or [N,1]): range vector for tauTab lookup in [m]

tauTab (np.array[N,] or [N,1]): transmittance vector for lookup in [m]

rangeGuess (float): starting value range estimate in [m] (optional)

n (float): range power (2 or 4) (optional)

Returns:

range (float or np.array[N,] or [N,1]): Solution to the range equation in [m]. Value is negative if calculated range exceeds the top value in range table, or if calculated range is too near the lower resolution limit.

Raises:

No exception is raised.

pyradi.ryutils.**_rangeEquationCalc**(*r*, *i*, *e*, *tauTable*, *n*, *rMax*)

pyradi.ryutils.**detectThresholdToNoiseTpFAR**(*pulseWidth*, *FAR*)

Solve for threshold to noise ratio, given pulse width and FAR, for matched filter.

Using the theory of matched filter design, calculate the threshold to noise ratio, to achieve a required false alarm rate.

References:

"Electro-optics handbook," Tech. Rep. EOH-11, RCA, 1974. RCA Technical Series Publication.

18. (a)Hippenstiel, Detection Theory: Applications and Digital Signal Pro-cessing, CRC Press, 2002

Args:

pulseWidth (float): the signal pulse width in [s].

FAR (float): the false alarm rate in [alarms/s]

Returns:

range (float): threshold to noise ratio

Raises:

No exception is raised.

pyradi.ryutils.**detectSignalToNoiseThresholdToNoisePd**(*ThresholdToNoise*, *pD*)

Solve for the signal to noise ratio, given the threshold to noise ratio and probability of detection.

Using the theory of matched filter design, calculate the signal to noise ratio, to achieve a required probability of detection.

References:

"Electro-optics handbook," Tech. Rep. EOH-11, RCA, 1974. RCA Technical Series Publication.

18. (a)Hippenstiel, Detection Theory: Applications and Digital Signal Pro-cessing, CRC Press, 2002

Args:

ThresholdToNoise (float): the threshold to noise ratio [-]

pD (float): the probability of detection [-]

Returns:

range (float): signal to noise ratio

Raises:

No exception is raised.

pyradi.ryutils.**detectThresholdToNoiseSignalToNoisepD**(*SignalToNoise*, *pD*)

Solve for the threshold to noise ratio, given the signal to noise ratio and probability of detection.

References:

"Electro-optics handbook," Tech. Rep. EOH-11, RCA, 1974. RCA Technical Series Publication.

18. (a)Hippenstiel, Detection Theory: Applications and Digital Signal Pro-cessing, CRC Press, 2002

Args:

SignalToNoise (float): the signal to noise ratio [-]
pD (float): the probability of detection [-]

Returns:

range (float): signal to noise ratio

Raises:

No exception is raised.

pyradi.ryutils.**detectProbabilityThresholdToNoiseSignalToNoise**(*ThresholdToNoise*, *SignalToNoise*)

Solve for the probability of detection, given the signal to noise ratio and threshold to noise ratio

References:

"Electro-optics handbook," Tech. Rep. EOH-11, RCA, 1974. RCA Technical Series Publication.

18. (a)Hippenstiel, Detection Theory: Applications and Digital Signal Pro-cessing, CRC Press, 2002

Args:

ThresholdToNoise (float): the threshold to noise ratio [-]
SignalToNoise (float): the signal to noise ratio [-]

Returns:

range (float): probability of detection

Raises:

No exception is raised.

pyradi.ryutils.**detectFARThresholdToNoisepulseWidth**(*ThresholdToNoise*, *pulseWidth*)

Solve for the FAR, given the threshold to noise ratio and pulse width, for matched filter.

References:

"Electro-optics handbook," Tech. Rep. EOH-11, RCA, 1974. RCA Technical Series Publication.

18. (a)Hippenstiel, Detection Theory: Applications and Digital Signal Pro-cessing, CRC Press, 2002

Args:

ThresholdToNoise (float): the threshold to noise ratio.
pulseWidth (float): the signal pulse width in [s].

Returns:

FAR (float): the false alarm rate in [alarms/s]

Raises:

No exception is raised.

pyradi.ryutils.**upMu**(*uprightMu=True*, *textcomp=False*)

Returns a LaTeX micron symbol, either an upright version or the normal symbol.

The upright symbol requires that the siunitx LaTeX package be installed on the computer running the code. This function also changes the Matplotlib rcParams file.

Args:

uprightMu (bool): signals upright (True) or regular (False) symbol (optional).

textcomp (bool): if True use the textcomp package, else use siunitx package (optional).

Returns:

range (string): LaTeX code for the micro symbol.

Raises:

No exception is raised.

pyradi.ryutils.**cart2polar**(*x*, *y*)

Converts from cartesian to polar coordinates, given (x,y) to (r,theta).

Args:

x (float np.array): x values in array format.

y (float np.array): y values in array format.

Returns:

r (float np.array): radial component for given (x,y).

theta (float np.array): angular component for given (x,y).

Raises:

No exception is raised.

original code by Joe Kington https://stackoverflow.com/questions/3798333/image-information-along-a-polar-coordinate-system

pyradi.ryutils.**polar2cart**(*r*, *theta*)

Converts from polar to cartesian coordinates, given (r,theta) to (x,y).

Args:

r (float np.array): radial values in array format.

theta (float np.array): angular values in array format.

Returns:

x (float np.array): x component for given (r, theta).

y (float np.array): y component for given (r, theta).

Raises:

No exception is raised.

original code by Joe Kington https://stackoverflow.com/questions/3798333/image-information-along-a-polar-coordinate-system

pyradi.ryutils.**index_coords**(*data*, *origin=None*, *framesFirst=True*)

Creates (x,y) zero-based coordinate arrrays for a numpy array indices, relative to some origin.

This function calculates two meshgrid arrays containing the coordinates of the input array. The origin of the new coordinate system defaults to the center of the image, unless the user supplies a new origin.

The data format can be data.shape = (rows, cols, frames) or data.shape = (frames, rows, cols), the format of which is indicated by the framesFirst parameter.

Args:

data (np.array): array for which coordinates must be calculated.

origin ((x-orig, y-orig)): data-coordinates of where origin should be

framesFirst (bool): True if data.shape is (frames, rows, cols), False if data.shape is (rows, cols, frames)

Returns:

x (float np.array): x coordinates in array format.

y (float np.array): y coordinates in array format.

Raises:

No exception is raised.

original code by Joe Kington https://stackoverflow.com/questions/3798333/image-information-along-a-polar-coordinate-system

pyradi.ryutils.**framesFirst**(*imageSequence*)

Image sequence with frames along axis=2 (last index), reordered such that frames are along axis=0 (first index).

Image sequences are stored in three-dimensional arrays, in rows, columns and frames. Not all libraries share the same sequencing, some store frames along axis=0 and others store frames along axis=2. This function reorders an image sequence with frames along axis=2 to an image sequence with frames along axis=0. The function uses np.transpose(imageSequence, (2,0,1))

Args:

imageSequence (3-D np.array): image sequence in three-dimensional array, frames along axis=2

Returns:

((3-D np.array): reordered three-dimensional array (view or copy)

Raises:

No exception is raised.

pyradi.ryutils.**framesLast**(*imageSequence*)

Image sequence with frames along axis=0 (first index), reordered such that frames are along axis=2 (last index).

Image sequences are stored in three-dimensional arrays, in rows, columns and frames. Not all libraries share the same sequencing, some store frames along axis=0 and others store frames along axis=2. This function reorders an image sequence with frames along axis=0 to an image sequence with frames along axis=2. The function uses np.transpose(imageSequence, (1,2,0))

Args:

imageSequence (3-D np.array): image sequence in three-dimensional array, frames along axis=0

Returns:

((3-D np.array): reordered three-dimensional array (view or copy)

Raises:

No exception is raised.

pyradi.ryutils.**rect**(*x*, *y*, *sx=1*, *sy=1*)

Generation of a rectangular aperture.

Args:

x (np.array[N,M]): x-grid, metres

y (np.array[N,M]): x-grid, metres
sx (float): full size along x.
sy (float): full size along y.

Returns:

Nothing.

Raises:

No exception is raised.

Author: CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.ryutils.**circ**(*x*, *y*, *d=1*)
Generation of a circular aperture.

Args:

x (np.array[N,M]): x-grid, metres
y (np.array[N,M]): y-grid, metres
d (float): diameter in metres.
comment (string): the symbol used to comment out lines, default value is None.
delimiter (string): delimiter used to separate columns, default is whitespace.

Returns:

z (np.array[N,M]): z-grid, 1's inside radius, meters/pixels.

Raises:

No exception is raised.

Author: Prof. Jason Schmidt, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.ryutils.**poissonarray**(*inp*, *seedval=None*, *tpoint=1000*)
This routine calculates a Poisson random variable for an array of input values with potentially very high event counts.

At high mean values the Poisson distribution calculation overflows. For mean values exceeding 1000, the Poisson distribution may be approximated by a Gaussian distribution.

The function accepts a two-dimensional array and calculate a separate random value for each element in the array, using the element value as the mean value. A typical use case is when calculating shot noise for image data.

From http://en.wikipedia.org/wiki/Poisson_distribution#Related_distributions For sufficiently large values of λ, (say $\lambda > 1000$), the normal distribution with mean λ and variance λ (standard deviation $\sqrt{\lambda}$) is an excellent approximation to the Poisson distribution. If λ is greater than about 10, then the normal distribution is a good approximation if an appropriate continuity correction is performed, i.e., $P(X \leq x)$, where (lower-case) x is a non-negative integer, is replaced by $P(X \leq x + 0.5)$.

$F_{\mathrm{Poisson}}(x;\lambda) \approx F_{\mathrm{normal}}(x;\mu=\lambda,\sigma^2=\lambda)$

Args:

inp (np.array[N,M]): array with mean value
seedval (int): seed for random number generator, None means use system time.
tpoint (int): Threshold when to switch over between Poisson and Normal distributions

Returns:

outp (np.array[N,M]): Poisson random variable for given mean value

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryutils.**draw_siemens_star**(*outfile*, *n*, *dpi*)

Siemens star chart generator

by Libor Wagner, http://cmp.felk.cvut.cz/~wagnelib/utils/star.html

Args:

outfile (str): output image filename (monochrome only)
n (int): number of spokes in the output image.
dpi (int): dpi in output image, determines output image size.

Returns:

Nothing, creates a monochrome siemens star image

Raises:

No exception is raised.

Author: Libor Wagner, adapted by CJ Willers

pyradi.ryutils.**gen_siemens_star**(*origin*, *radius*, *n*)

pyradi.ryutils.**drawCheckerboard**(*rows*, *cols*, *numPixInBlock*, *imageMode*, *colour1*, *colour2*, *imageReturnType='image'*)

Draw checkerboard with 8-bit pixels

From http://stackoverflow.com/questions/2169478/how-to-make-a-checkerboard-in-numpy

Args:

rows : number or rows in checkerboard
cols : number of columns in checkerboard
numPixInBlock : number of pixels to be used in one block of the checkerboard
imageMode : PIL image mode [e.g. L (8-bit pixels, black and white), RGB (3x8-bit pixels, true color)]
colour1 : colour 1 specified according to the imageMode
colour2 : colour 2 specified according to the imageMode
imageReturnType: 'image' for PIL image, 'nparray' for numpy array

Returns:

img : checkerboard numpy array or PIL image (see imageReturnType)

Raises:

No exception is raised.

Example Usage:

rows = 5 cols = 7 pixInBlock = 4

color1 = 0 color2 = 255 img = drawCheckerboard(rows,cols,pixInBlock,'L',color1,color2,'nparray') pilImg = Img.fromarray(img, 'L') pilImg.save('{0}.png'.format('checkerboardL'))

color1 = (0,0,0) color2 = (255,255,255) pilImage = drawCheckerboard(rows,cols,pixInBlock,'RGB',color1,color2,'image') pilImage.save('{0}.png'.format('checkerboardRGB'))

pyradi.ryutils.**makemotionsequence**(*imgfilename*, *mtnfilename*, *postfix*, *intTime*, *frmTim*, *outrows*, *outcols*, *imgRatio*, *pixsize*, *numsamples*, *fnPlotInput=None*)

Builds a video from a still image and a displacement motion file.

The objective with this function is to create a video sequence from a still image, as if the camera moved minutely during the sensor integration time.

A static image is moved according to the (x,y) displacement motion in an input file. The input file must be at least ten times plus a bit larger than the required output file. The image input file is sampled with appropriate displacement for each point in the displacement file and pixel vlaues are accumulated in the output image. All of this temporal displacement and accumulation takes place in the context of a frame integration time and frame frequency.

The key requirements for accuracy in this method is an input image with much higher resolution than the output image, plus a temporal displacement file with much higher temporal sampling than the sensor integration time.

The function creates a sequence of images that can be used to create a video. Images are numbered in sequence, using the same base name as the input image. The sequence is generated in the current working directory.

The function currently processes only monochrome images (M,N) arrays.

The motion data file must be a compressed numpy npz or text file, with three columns: First column must be time, then movement along rows, then movement along columns. The units and scale of the motion columns must be the same units and scale as the pixel size in the output image.

imgRatio x imgRatio number of pixels in the input (hires) image are summed together and stored in one output image pixel. In other words if imgRatio is ten, each pixel in the output image will be the sum of 100 pixels in the imput image. During one integration time period the hires input image will be sampled at slightly different offsets (according to the motion file) and accumulated in an intermediate internal hires file. This intermediate internal file is collapsed as described above.

The function creates a series-numbered sequence if images that can be used to construct a video. One easy means to create the video is to use VirtualDub, available at www.virtualdub.org/index. In VirtualDub open the first image file in the numbered sequence, VirtualDub will then recognise the complete sequence as a video. Once loaded in VirtualDub, save the video as avi.

Args:

imgfilename (str): static image filename (monochrome only)
mtnfilename (str): motion data filename.
postfix (str): add this string to the end of the output filename.
intTime (float): sensor integration time.
frmTim (float): sensor frame time.
outrows (int): number of rows in the output image.
outcols (int): number of columns in the output image.
imgRatio (float): hires image pixel count block size of one output image pixel
pixsize (float): pixel size in same units as motion file.
numsamples (int): number of motion input samples to be processed (-1 for all).
fnPlotInput (str): output plot filename (None for no plot).

Returns:

True if successful, message otherwise, creates numbered images in current working directory

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryutils.**extractGraph**(*filename*, *xmin*, *xmax*, *ymin*, *ymax*, *outfile=None*, *doPlot=False*, *xaxisLog=False*, *yaxisLog=False*, *step=None*, *value=None*)

Scan an image containing graph lines and produce (x,y,value) data.

This function processes an image, calculate the location of pixels on a graph line, and then scale the (r,c) or (x,y) values of pixels with non-zero values. The

Get a bitmap of the graph (scan or screen capture). Take care to make the graph x and y axes horizontal/vertical. The current version of the software does not work with rotated images. Bitmap edit the graph. Clean the graph to the maximum extent possible, by removing all the clutter, such that only the line to be scanned is visible. Crop only the central block that contains the graph box, by deleting the x and y axes notation and other clutter. The size of the cropped image must cover the range in x and y values you want to cover in the scan. The graph image/box must be cut out such that the x and y axes min and max correspond exactly with the edges of the bitmap. You must end up with nothing in the image except the line you want to digitize.

The current version only handles single lines on the graph, but it does handle vertical and horizontal lines.

The function can also write out a value associated with the (x,y) coordinates of the graph, as the third column. Normally these would have all the same value if the line represents an iso value.

The x,y axes can be lin/lin, lin/log, log/lin or log/log, set the flags.

Args:

filename: name of the image file
xmin: the value corresponding to the left side (column=0)
xmax: the value corresponding to the right side (column=max)
ymin: the value corresponding to the bottom side (row=bottom)
ymax: the value corresponding to the top side (row=top)
outfile: write the sampled points to this output file
doPlot: plot the digitised graph for visual validation
xaxisLog: x-axis is in log10 scale (min max are log values)
yaxisLog: y-axis is in log10 scale (min max are log values)
step: if not None only ouput every step values
value: if not None, write this value as the value column

Returns:

outA: a numpy array with columns (xval, yval, value)
side effect: a file may be written
side effect: a graph may be displayed

Raises:

No exception is raised.

Author: neliswillers@gmail.com

pyradi.ryutils.**luminousEfficiency**(*vlamtype='photopic'*, *wavelen=None*, *eqnapprox=False*)

Returns the photopic luminous efficiency function on wavelength intervals

Type must be one of:

photopic: CIE Photopic V(lambda) modified by Judd (1951) and Vos (1978) [also known as CIE VM(lambda)] scotopic: CIE (1951) Scotopic V'(lambda) CIE2008v2: 2 degree CIE "physiologically-relevant" luminous efficiency Stockman & Sharpe CIE2008v10: 10 degree CIE "physiologically-relevant" luminous efficiency Stockman & Sharpe

For the equation approximations (only photoic and scotopic), if wavelength is not given a vector is created 0.3-0.8 um.

For the table data, if wavelength is not given a vector is read from the table.

CIE Photopic V(l) modified by Judd (1951) and Vos (1978) [also known as CIE VM(l)] from http://www.cvrl.org/index.htm

Args:

vlamtype (str): type of curve required
wavelen (np.array[]): wavelength in um

eqnapprox (bool): if False read tables, if True use equation

Returns:

luminousEfficiency (np.array[]): luminous efficiency

wavelen (np.array[]): wavelength in um

Raises:

No exception is raised.

Author: CJ Willers

pyradi.ryutils.**calcMTFwavefrontError**(*sample*, *wfdisplmnt*, *xg*, *yg*, *specdef*, *samplingStride=1*, *clear='Clear'*)

Given a mirror figure error, calculate MTF degradation from ideal

An aperture has an MTF determined by its shape. A clear aperture has zero phase delay and the MTF is determined only by the aperture shape. Any phase delay/error in the wavefront in the aperture will result in a lower MTF than the clear aperture diffraction MTF.

This function calculates the MTF degradation attributable to a wavefront error, relative to the ideal aperture MTF.

The optical transfer function is the Fourier transform of the point spread function, and the point spread function is the square absolute of the inverse Fourier transformed pupil function. The optical transfer function can also be calculated directly from the pupil function. From the convolution theorem it can be seen that the optical transfer function is the autocorrelation of the pupil function <https://en.wikipedia.org/wiki/Optical_transfer_function>.

The pupil function comprises a masking shape (the binary shape of the pupil) and a transmittance and spatial phase delay inside the mask. A perfect aperture has unity transmittance and zero phase delay in the mask. Some pupils have irregular pupil functions/shapes and hence the diffraction MTF has to be calculated numerically using images (masks) of the pupil function.

From the OSA Handbook of Optics, Vol II, p 32.4: For an incoherent optical system, the OTF is proportional to the two-dimensional autocorrelation of the exit pupil. This calculation can account for any phase factors across the pupil, such as those arising from aberrations or defocus. A change of variables is required for the identification of an autocorrelation (a function of position in the pupil) as a transfer function (a function of image-plane spatial frequency). The change of variables is

xi = {x}/{lambda d_i}

where x is the autocorrelation shift distance in the pupil, $lambda$ is the wavelength, and d_i is the distance from the exit pupil to the image. A system with an exit pupil of full width D has an image-space cutoff frequency (at infinite conjugates) of

xi_{cutoff} ={D}/{lambda f}

In this analysis we assume that 1. the sensor is operating at infinite conjugates. 2. the mask falls in the entrance pupil shape.

The MTF is calculated as follows:

1. Read in the pupil function mask and create an image of the mask.
2. Calculate the two-dimensional autocorrelation function of the binary image (using the SciPy two-dimensional correlation function *signal.correlate2d*).
3. Scale the magnitude and (x,y) dimensions according to the dimensions of the physical pupil.

The the array containing the wavefront displacement in the pupil must have np.nan values outside the pupil. The np.nan values are ignored and not included in the calculation. Obscurations can be modelled by placing np.nan in the obscuration.

The specdef dictionary has a string key to identify (name) the band, with a single float contents which is the wavelength associated with this band.

Args:

sample (string): an identifier string to be used in the plots
wfdisplmnt (nd.array[M,N]): wavefront displacement in m
xg (nd.array[M,N]): x values from meshgrid, for wfdisplmnt
yg (nd.array[M,N]): y values from meshgrid, for wfdisplmnt
specdef (dict): dictionary defining spectral wavelengths
samplingStride (number): sampling stride to limit size and processing
clear (string): defines the dict key for clear aperture reference

Returns:

dictionaries below have entries for all keys in specdef.
wfdev (dict): subsampled wavefront error in m
phase (dict): subsampled wavefront error in rad
pupfn (dict): subsampled complex pupil function
MTF2D (dict): 2D MTF in (x,y) format
MTFpol (dict): 2D MTF in (r,theta) format
specdef (): specdef dictionary as passed plus clear entry
MTFmean (dict): mean MTF across all rotation angles
rho (nd.array[M,]): spatial frequency scale in cy/mrad
fcrit (float): cutoff or critical spatial frequency cy/mrad
clear (string): key used to signify the clear aperture case.

Raises:

No exception is raised.

CHAPTER

SEVEN

RADIOMETRIC LOOKUP FUNCTIONS

7.1 Overview

This class provides lookup functionality between source temperature, source radiance and sensor signal. In this class the sensor signal is called digital level, but it represents any atribrary sensor signal unit.

See the __main__ function for examples of use.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

7.2 Module classes

class `pyradi.rylookup.`**`RadLookup`**(*specName*, *nu*, *tmprLow*, *tmprHi*, *tmprInc*, *sensorResp=None*, *opticsTau=None*, *filterTau=None*, *atmoTau=None*, *sourceEmis=None*, *sigMin=None*, *sigMax=None*, *sigInc=None*, *dicCaldata=None*, *dicPower=None*, *dicFloor=None*)

Performs radiometric lookup capability between temperature and radiance, given camera spectral and calibration data.

Given spectral data and temperature this class calculates loopup tables and provide lookup functions operating on the tables.

The class provides two parallel functional capabilities:

- given spectral data and no calibration data it calculates a simple radiance-based lookup between temperture and radiance, assuming Planck-law relationships.
- given camera calibration data it relates between signal value, temperature and radiance. It accounts for the effect of hot optics that cause a lower asymptotic radiance level. The calibration mode supports linear interpolation between two calibration curves, to account for the instrument internal temperature. This mode requires the sigMin, sigMax, sigInc, and dicCaldat parameters.

By not passing the calibration parameters simply means that that part of the code is not executed and only the simple radiance-based lookup is available.

Five spectral vectors can be supplied: (1) emissivity, (2) atmospheric transmittance, (3) filter transmittance, (4) optics tranmittance, and (5) sensor response. Two sets of calculations are performed, the first with all five spectral vectors, and the second without the filter, but all four remaining vectors. This option may be useful when compensating for neutral density filters, i.e., radiance before or after the filter (with/without). This option is relevant only in the functions `LookupRadTemp`, `LookupTempRad`, and `PlotTempRadiance`. In these three functions, the appropriate option can be selected by setting the `withFilter=True` function parameter.

Spectral data parameter may be either a filename (data read from file) or a numpy array np.array(:,1) with the data on (nuMin, nuMax,nuInc) scale. The data file must have two colums: first column is wavelength, and second column is the spectral value at this wavelength. Data read in from the file will be interpolated to (nuMin, nuMax,nuInc) scale. If the parameter is None, then unity spectral values will be used.

The camera calibration data set requires the following data:

- sigMin, sigMax, sigInc: the parameters to define the signal magnitude vector.
- dicCaldata: the dictionary containing the camera calibration data. The dictionary key is the instrument internal temperature [deg-C] (one or two values required). For each key, provide a numpy array where the *first column* is the calibration source temperature K, and the *second column* is the signal measured on the instrument and the *third column* is the radiance for this temperature (only after the tables have been calculated).
- dicPower: the dictionary containing the camera calibration curve lower knee sharpness. The dictionary key is the instrument internal temperature (one or two values required).
- dicFloor: the dictionary containing the camera calibration curve lower asymptotic signal value. The dictionary key is the instrument internal temperature (one or two values required).

Error handling is simply to test for certain conditions and then execute the task if conditions are met, otherwise do nothing.

Args:

specName (string): Name for this lookup data set, used in graphs.
nu (np.array(N,1)): wavenumber vector to be used in spectral calcs.
tmprLow (float): Lower temperature bound [K] for lookup tables.
tmprHi (float): Upper temperature bound [K] for lookup tables.
tmprInc (float): Increment temperature [K] for lookup tables.
sensorResp (string/np.array(N,1)): sensor/detector spectral filename or array data (optional).
opticsTau (string/np.array(N,1)): opticsTransmittance spectral filename or array data (optional).
filterTau ((string/np.array(N,1)): filter spectral filename or array data (optional).
atmoTau (string/np.array(N,1)): atmoTau spectral filename or array data (optional).
sourceEmis (string/np.array(N,1)): sourceEmis spectral filename or array data (optional).
sigMin (float): minimum signal value, typically 2**0 (optional).
sigMax (float): maximum signal value, typically 2**14 (optional).
sigInc(float): signal increment, typically 2**8 (optional).
dicCaldata (dict): calibration data for sensor.
dicPower (dict): cal curve lower asymptote knee sharpness parameter (optional).
dicFloor (dict): cal curve lower asymptote parameter (optional).

Returns:

Nothing, but init() loads data and calculates the tables on instantiation.

Raises:

No exception is raised.

`CalculateDataTables`()

Calculate the mapping functions between sensor signal, radiance and temperature.

Using the spectral curves and DL vs. temperature calibration inputs calculate the various mapping functions between digital level, radiance and temperature. Set up the various tables for later conversion.

The TableTempRad table has three columns: (1) temperature, (2) radiance with filter present and (3) radiance without filter.

Args:

None.

Returns:

None. Side effect of all tables calculated.

Raises:

No exception is raised.

Info()

Write the calibration data file data to a string and return string.

Args:

None.

Returns:

(string) containing the key information for this class.

Raises:

No exception is raised.

LoadSpectrals()

Load the five required spectral parameters, interpolate on the fly to local spectrals.

If the spectral parameters are strings, the strings are used as filenames and the data loaded from file. If None, unity values are assumed. If not a string or None, the parameters are used as is, and must be numpy arrays with shape (N,1) where the N vector exactly matches to spectral samples

Args:

None.

Returns:

None. Side-effect of loaded spectrals.

Raises:

No exception is raised.

LookupDLRad(*DL*, *Tint*)

Calculate the radiance associated with a DL and Tint pair. Interpolate linearly on Tint radiance not temperature.

Args:

DL (float, np.array[N,]): scalar, list or array of sensor signal values.
Tint (float): scalar, internal temperature of the sensor.

Returns:

(np.array[N,]) radiance W/(sr.m2) values associated with sensor signals.

Raises:

No exception is raised.

LookupDLRadHelper(*DL*, *paraK*)

Calculate the radiance associated with a DL and parametric pair. The parametric variable was calculated once and used for all DL values.

Args:

DL (float, np.array[N,]): scalar, list or array of sensor signal values.
paraK (float): scalar, internal temperature parametric values.

Returns:

(np.array[N,]) radiance W/(sr.m2) values associated with sensor signals.

Raises:

No exception is raised.

LookupDLTemp(*DL*, *Tint*)

Calculate the temperature associated with a DL and Tint pair. Interpolate linearly on Tint temperature - actually we must interpolate linearly on radiance - to be done later.

Args:

DL (float, np.array[N,]): scalar, list or array of sensor signal values.
Tint (float): scalar, internal temperature of the sensor.

Returns:

(np.array[N,]) temperature K values associated with sensor signals.

Raises:

No exception is raised.

LookupRadTemp(*radiance*, *withFilter=True*)

Calculate the temperature associated with a radiance value for the given spectral shapes (no calibration involved).

Args:

radiance (float, np.array[N,]): scalar, list or array of radiance W/(sr.m2) values.
withFilter (boolean): use table with filter to do lookup, no filter if false (optional).

Returns:

(np.array[N,]) temperature K values associated with radiance values.

Raises:

No exception is raised.

LookupTempRad(*temperature*, *withFilter=True*)

Calculate the radiance associated with a temperature for the given spectral shapes (no calibration involved).

Args:

temperature(np.array[N,]) scalar, list or array of temperature K values.
withFilter (boolean): use table with filter to do lookup, no filter if false (optional).

Returns:

radiance (float, np.array[N,]): radiance W/(sr.m2) associated with temperature values..

Raises:

No exception is raised.

PlotCalDLRadiance(*savePath=None*, *saveExt='png'*)

Plot DL level versus radiance for both camera temperatures

The filename is constructed from the given object name, save path, and the word 'CaldlRadiance'.

Args:

savePath (string): Path to where the plots must be saved (optional).
saveExt (string) : Extension to save the plot as, default of 'png' (optional).

Returns:

None, the images are saved to a specified location or in the location
from which the script is running.

Raises:

No exception is raised.

PlotCalDLTemp(*savePath=None*, *saveExt='png'*)
Plot digital level versus temperature for both camera temperatures

The filename is constructed from the given object name, save path, and the word 'CalDLTemp'.

Args:

savePath (string): Path to where the plots must be saved (optional).
saveExt (string) : Extension to save the plot as, default of 'png' (optional).

Returns:

None, the images are saved to a specified location or in the location
from which the script is running.

Raises:

No exception is raised.

PlotCalSpecRadiance(*savePath=None*, *saveExt='png'*)
Plot spectral radiance data for the calibration temperatures.

The filename is constructed from the given object name, save path, and the word 'CalRadiance'.

Args:

savePath (string): Path to where the plots must be saved (optional).
saveExt (string) : Extension to save the plot as, default of 'png' (optional).

Returns:

None, the images are saved to a specified location or in the location
from which the script is running.

Raises:

No exception is raised.

PlotCalTempDL(*savePath=None*, *saveExt='png'*)
Plot digital level versus temperature for both camera temperatures

The filename is constructed from the given object name, save path, and the word 'CalDLTemp'.

Args:

savePath (string): Path to where the plots must be saved (optional).
saveExt (string) : Extension to save the plot as, default of 'png' (optional).

Returns:

None, the images are saved to a specified location or in the location
from which the script is running.

Raises:

No exception is raised.

PlotCalTempRadiance(*savePath=None*, *saveExt='png'*)
Plot temperature versus radiance for both camera temperatures

The filename is constructed from the given object name, save path, and the word 'CalTempRadiance'.

Args:

savePath (string): Path to where the plots must be saved (optional).
saveExt (string) : Extension to save the plot as, default of 'png' (optional).

Returns:

None, the images are saved to a specified location or in the location

from which the script is running.

Raises:

No exception is raised.

PlotCalTintRad (*savePath=None*, *saveExt='png'*)
Plot optics radiance versus instrument temperature

The filename is constructed from the given object name, save path, and the word 'CalInternal'.

Args:

savePath (string): Path to where the plots must be saved (optional).
saveExt (string) : Extension to save the plot as, default of 'png' (optional).

Returns:

None, the images are saved to a specified location or in the location
from which the script is running.

Raises:

No exception is raised.

PlotSpectrals (*savePath=None*, *saveExt='png'*)
Plot all spectral curve data to a single graph.

The filename is constructed from the given object name, save path, and the word 'spectrals'.

Args:

savePath (string): Path to where the plots must be saved (optional).
saveExt (string) : Extension to save the plot as, default of 'png' (optional).

Returns:

None, the images are saved to a specified location or in the location
from which the script is running.

Raises:

No exception is raised.

PlotTempRadiance (*withFilter=True*, *savePath=None*, *saveExt='png'*)
Plot temperature versus radiance for both camera temperatures

The filename is constructed from the given object name, save path, and the word 'TempRadiance'.

Args:

withFilter (boolean): use table with filter to do lookup, no filter if false (optional).
savePath (string): Path to where the plots must be saved (optional).
saveExt (string) : Extension to save the plot as, default of 'png' (optional).

Returns:

None, the images are saved to a specified location or in the location
from which the script is running.

Raises:

No exception is raised.

CHAPTER

EIGHT

PTW FILE FUNCTIONS

8.1 Overview

This module provides functionality to read the contents of files in the PTW file format and convert the raw data to source radiance or souorce temperature (provided that the instrument calibration data is available).

Functions are provided to read the binary Agema/Cedip/FLIR Inc PTW format into data structures for further processing.

The following functions are available to read PTW files:

readPTWHeader(ptwfilename)

showHeader(header)

getPTWFrame (header, frameindex)

readPTWHeader(ptwfilename) : Returns a class object defining all the header information in ptw file.

showHeader(header) : Returns nothing. Prints the PTW header content to the screen.

getPTWFrame (header, frameindex) : Return the raw DL levels of the frame defined by frameindex.

The authors wish to thank FLIR Advanced Thermal Solutions for the permission to publicly release our Python version of the ptw file reader. Please note that the copyright to the proprietary ptw file format remains the property of FLIR Inc.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

8.2 Module classes

class pyradi.ryptw.**JadeCalibrationData**(*filename*, *datafileroot*)

Container to describe the calibration data of a Jade camera.

Info()

Write the calibration data file data to a string and return string.

class pyradi.ryptw.**PTWFrameInfo**

Class to store the ptw file header information.

8.3 Module functions

pyradi.ryptw.**readPTWHeader**(*ptwfilename*)

Given a ptw filename, read the header and return the header to caller

Args:

filename (string) with full path to the ptw file.

Returns:

Header (class) containing all PTW header information.

Raises:

No exception is raised.

Reference: h_variables of the header and byte positions are obtained from DL002U-D Altair Reference Guide

pyradi.ryptw.**showHeader**(*Header*)

Utility function to print the PTW header information to stdout

Args:

header (class object) ptw file header structure

Returns:

None

Raises:

No exception is raised.

pyradi.ryptw.**getPTWFrame**(*header*, *frameindex*)

Retrieve a single PTW frame, given the header and frame index

This routine also stores the data array as part of the header. This may change - not really needed to have both a return value and header stored value for the DL valueheader. This for a historical reason due to the way GetPTWFrameFromFile was written

Args:

header (class object)
frameindex (integer): The frame to be extracted

Returns:

header.data (np.ndarray): requested frame DL values, dimensions (rows,cols)

Raises:

No exception is raised.

pyradi.ryptw.**GetPTWFrameFromFile**(*header*)

From the ptw file, load the frame specified in the header variable header.h_framepointer

Args:

header (class object) header of the ptw file, with framepointer set

Returns:

header.data plus newly added information: requested frame DL values, dimensions (rows,cols)

Raises:

No exception is raised.

pyradi.ryptw.**getPTWFrames**(*header*, *loadFrames=[]*)

Retrieve a number of PTW frames, given in a list of frameheader.

Args:

header (class object)
loadFrames ([int]): List of indices for frames to be extracted

Returns:

data (np.ndarray): requested image frame DL values, dimensions (frames,rows,cols)

Raises:

No exception is raised.

pyradi.ryptw.**myint**(*x*)

pyradi.ryptw.**mylong**(*x*)

pyradi.ryptw.**myfloat**(*x*)

pyradi.ryptw.**mybyte**(*x*)

pyradi.ryptw.**mydouble**(*x*)

pyradi.ryptw.**terminateStrOnZero**(*str*)

Iterate through string and terminate on first zero

CHAPTER

NINE

MODTRAN UTILITY

9.1 Overview

This module provides MODTRAN tape7 file reading. Future development may include a class to write tape5 files, but that is a distant target at present.

See the __main__ function for examples of use.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

9.2 Module classes

pyradi.rymodtran.**fixHeaders**(*instr*)

Modifies the column header string to be compatible with numpy column lookup.

Args:

list columns (string): column name.

Returns:

list columns (string): fixed column name.

Raises:

No exception is raised.

pyradi.rymodtran.**loadtape7**(*filename*, *colspec=[]*)

Read the Modtran tape7 file. This function was tested with Modtran5 files.

Args:

filename (string): name of the input ASCII flatfile.

colspec ([string]): list of column names required in the output the spectral transmittance data.

Returns:

np.array: an array with the selected columns. Col[0] is the wavenumber.

Raises:

No exception is raised.

This function reads in the tape7 file from MODerate spectral resolution atmospheric TRANsmission (MODTRAN) code, that is used to model the propagation of the electromagnetic radiation through the atmosphere. tape7 is a primary file that contains all the spectral results of the MODTRAN run. The header information in the tape7 file contains portions of the tape5 information that will be deleted. The header section in

tape7 is followed by a list of spectral points with corresponding transmissions. Each column has a different component of the transmission or radiance. For more detail, see the modtran documentation.

The user selects the appropriate columns by listing the column names, as listed below.

The format of the tape7 file changes for different IEMSCT values. For the most part the differences are hidden in the details. The various column headers used in the tape7 file are as follows:

IEMSCT = 0 has two column header lines. Different versions of modtran has different numbers of columns. In order to select the column, you must concatenate the two column headers with an underscore in between. All columns are available with the following column names: ['FREQ_CM-1', 'COMBIN_TRANS', 'H2O_TRANS', 'UMIX_TRANS', 'O3_TRANS', 'TRACE_TRANS', 'N2_CONT', 'H2O_CONT', 'MOLEC_SCAT', 'AER+CLD_TRANS', 'HNO3_TRANS', 'AER+CLD_abTRNS', '-LOG_COMBIN', 'CO2_TRANS', 'CO_TRANS', 'CH4_TRANS', 'N2O_TRANS', 'O2_TRANS', 'NH3_TRANS', 'NO_TRANS', 'NO2_TRANS', 'SO2_TRANS', 'CLOUD_TRANS', 'CFC11_TRANS', 'CFC12_TRANS', 'CFC13_TRANS', 'CFC14_TRANS', 'CFC22_TRANS', 'CFC113_TRANS', 'CFC114_TRANS', 'CFC115_TRANS', 'CLONO2_TRANS', 'HNO4_TRANS', 'CHCL2F_TRANS', 'CCL4_TRANS', 'N2O5_TRANS','H2-H2_TRANS','H2-HE_TRANS','H2-CH4_TRANS', 'CH4-CH4_TRANS']

IEMSCT = 1 has single line column headers. A number of columns have headers, but with no column numeric data. In the following list the columns with header names ** are empty and hence not available: ['FREQ', 'TOT_TRANS', 'PTH_THRML', 'THRML_SCT', 'SURF_EMIS', *SOL_SCAT*, *SING_SCAT*, 'GRND_RFLT', *DRCT_RFLT*, 'TOTAL_RAD', *REF_SOL*, *SOL@OBS*, 'DEPTH', 'DIR_EM', *TOA_SUN*, 'BBODY_T[K]']. Hence, these columns do not have valid data: ['SOL_SCAT', 'SING_SCAT', 'DRCT_RFLT', 'REF_SOL', 'SOL@OBS', 'TOA_SUN']

IEMSCT = 2 has single line column headers. All the columns are available: ['FREQ', 'TOT_TRANS', 'PTH_THRML', 'THRML_SCT', 'SURF_EMIS', 'SOL_SCAT', 'SING_SCAT', 'GRND_RFLT', 'DRCT_RFLT', 'TOTAL_RAD', 'REF_SOL', 'SOL@OBS', 'DEPTH', 'DIR_EM', 'TOA_SUN', 'BBODY_T[K]']

IEMSCT = 3 has single line column headers. One of these seems to be two words, which, in this code must be concatenated with an underscore. There is also additional column (assumed to be depth in this code). The columns available are ['FREQ', 'TRANS', 'SOL_TR', 'SOLAR', 'DEPTH']

The tape7.scn file has missing columns, so this function does not work for tape7.scn files. If you need a tape7.scn file with all the columns populated you would have to use the regular tape7 file and convolve this to lower resolution.

UPDATE: Different versions of Modtran have different columns present in the tape7 file, also not all the columns are necessarily filled, some have headings but no data. To further complicate matters, the headings are not always separated by spaces, sometime (not often) heading text runs into each other with no separator.

It seems that the column headings are right aligned with the column data itself, so if we locate the right most position of the column data, we can locate the headings with valid data - even in cases with connected headings.

The algorithm used is as follows:

1.step down to a data line beyond the heading lines (i.e., the data)

2.locate the position of the last character of every discrete columnn

3.From the last column of the previous col, find start of next col

4.move up into the header again and isolate the header column text for each column

In sptite of all the attempts to isolate special cases, reading of tape7 files remain a challange and may fail in newer versions of Modtran, until fixed.

pyradi.rymodtran.**fixHeadersList**(*headcol*)

Modifies the column headers to be compatible with numpy column lookup.

Args:

list columns ([string]): list of column names.

Returns:

list columns ([string]): fixed list of column names.

Raises:

No exception is raised.

pyradi.rymodtran.**runModtranAndCopy**(*root*, *research*, *pathToModtranBin*, *execname*)

Look for input files in directories, run modtran and copy results to dir.

Finds all files below the root directory that matches the regex pattern in research. Then runs modtran on these files and write the tape5/6/7/8 back to where the input file was.

Each input file must be in a separate directory, because the results are all written to files with the names 'tape5', etc.

Args:

root ([string]): path to root dir containing the dirs with modtran input files.

research ([string]): regex to use when searching for input files ('.*.ltn' or 'tape5')

pathToModtranBin ([string]): path to modtran executable directory.

execname ([string]): modtran executable filename.

Returns:

List of all the files processed.

Raises:

No exception is raised.

CHAPTER

TEN

THREE-DIMENSIONAL NOISE CALCULATION

10.1 Overview

This module provides a set of functions to aid in the calculation of 3D noise parameters from noise images. The functions are based on the work done by John D'Agostino and Curtis Webb. For details see "3-D Analysis Framwork and Measurement Methodology for Imaging System Nioise" p110-121 in "Infrared Imaging Systems: Design, Analysis, Modelling, and Testing II", Holst, G. C., ed., Volume 1488, SPIE (1991).

See the __main__ function for examples of use.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

10.2 Module functions

pyradi.ry3dnoise.**oprDT**(*imgSeq*)
Operator DT averages over frames for each pixel.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

numpy array of dimensions (1,rows,cols)

Raises:

No exception is raised.

pyradi.ry3dnoise.**oprDV**(*imgSeq*)
Operator DV averages over rows for each pixel.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

numpy array of dimensions (frames,1,cols)

Raises:

No exception is raised.

pyradi.ry3dnoise.**oprDH**(*imgSeq*)
Operator DH averages over columns for each pixel.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

numpy array of dimensions (frames,rows,1)

Raises:

No exception is raised.

pyradi.ry3dnoise.**oprSDT**(*imgSeq*)

Operator SDT first averages over frames for each pixel. The result is subtracted from all images.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

numpy array of dimensions (frames,rows,cols)

Raises:

No exception is raised.

pyradi.ry3dnoise.**oprSDV**(*imgSeq*)

Operator SDV first averages over rows for each pixel. The result is subtracted from all images.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

numpy array of dimensions (frames,rows,cols)

Raises:

No exception is raised.

pyradi.ry3dnoise.**oprSDH**(*imgSeq*)

Operator SDH first averages over columns for each pixel. The result is subtracted from all images.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

numpy array of dimensions (frames,rows,cols)

Raises:

No exception is raised.

pyradi.ry3dnoise.**getS**(*imgSeq*)

Average over all pixels.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): average of all pixels

Raises:

No exception is raised.

pyradi.ry3dnoise.**getNT**(*imgSeq*)

Average for all pixels as a function of time/frames. Represents noise which consists of fluctuations in the temporal direction affecting the mean of each frame.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): frame-to-frame intensity variation

Raises:

No exception is raised.

pyradi.ry3dnoise.**getNH**(*imgSeq*)

Average for each row over all frames and cols. Represents variations in column averages that are fixed in time.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): fixed column noise

Raises:

No exception is raised.

pyradi.ry3dnoise.**getNV**(*imgSeq*)

Average for each column over all frames and rows. Represents variations in row averages that are fixed in time.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): fixed row noise

Raises:

No exception is raised.

pyradi.ry3dnoise.**getNVH**(*imgSeq*)

Average over all frames, for each pixel. Represents non-uniformity spatial noise that does not change from frame-to-frame.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): fixed spatial noise

Raises:

No exception is raised.

pyradi.ry3dnoise.**getNTV**(*imgSeq*)

Average for each row and frame over all columns. Represents variations in row averages that change from frame-to-frame.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): row temporal noise

Raises:

No exception is raised.

pyradi.ry3dnoise.**getNTH**(*imgSeq*)

Average for each column and frame over all rows. Represents variations in column averages that change from frame-to-frame.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): column temporal noise

Raises:

No exception is raised.

pyradi.ry3dnoise.**getNTVH**(*imgSeq*)

Noise for each row, frame & column. Represents random noise in the detector and electronics.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): temporal pixel noise

Raises:

No exception is raised.

pyradi.ry3dnoise.**getTotal**(*imgSeq*)

Total system noise.

Args:

imgSeq (np.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): total system noise

Raises:

No exception is raised.

CHAPTER

ELEVEN

COLOUR COORDINATES

11.1 Overview

This module provides rudimentary colour coordinate processing. Calculate the CIE 1931 rgb chromaticity coordinates for an arbitrary spectrum.

See the __main__ function for examples of use.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

11.2 Module functions

pyradi.rychroma.**chromaticityforSpectralL**(*spectral*, *radiance*, *xbar*, *ybar*, *zbar*)

Calculate the CIE chromaticity coordinates for an arbitrary spectrum.

Given a spectral radiance vector and CIE tristimulus curves, calculate the CIE chromaticity coordinates. It is assumed that the radiance spectral density is given in the same units as the spectral vector (i.e. [1/um] or [1/cm-1], corresponding to [um] or [cm-1] respectively. It is furthermore accepted that the tristimulus curves are also sampled at the same spectral intervals as the radiance. See http://en.wikipedia.org/wiki/CIE_1931_color_space for more information on CIE tristimulus spectral curves.

Args:

spectral (np.array[N,] or [N,1]): spectral vector in [um] or [cm-1].

radiance (np.array[N,] or [N,1]): the spectral radiance (any units), (sampled at spectral).

xbar (np.array[N,] or [N,1]): CIE x tristimulus spectral curve (sampled at spectral values).

ybar (np.array[N,] or [N,1]): CIE y tristimulus spectral curve (sampled at spectral values).

zbar (np.array[N,] or [N,1]): CIE z tristimulus spectral curve (sampled at spectral values).

Returns:

[x,y,Y]: color coordinates x, y, and Y.

Raises:

No exception is raised.

pyradi.rychroma.**XYZforSpectralL**(*spectral*, *radiance*, *xbar*, *ybar*, *zbar*)

Calculate the CIE chromaticity coordinates for an arbitrary spectrum.

Given a spectral radiance vector and CIE tristimulus curves, calculate the XYZ chromaticity coordinates. It is assumed that the radiance spectral density is given in the same units as the spectral vector (i.e. [1/um] or [1/cm-1], corresponding to [um] or [cm-1] respectively. It is furthermore accepted that the tristimulus curves are also sampled at the same spectral intervals as the radiance. See http://en.wikipedia.org/wiki/CIE_1931_color_space for more information on CIE tristimulus spectral curves.

Args:

spectral (np.array[N,] or [N,1]): spectral vector in [um] or [cm-1].
radiance (np.array[N,] or [N,1]): the spectral radiance (any units), (sampled at spectral).
xbar (np.array[N,] or [N,1]): CIE x tristimulus spectral curve (sampled at spectral values).
ybar (np.array[N,] or [N,1]): CIE y tristimulus spectral curve (sampled at spectral values).
zbar (np.array[N,] or [N,1]): CIE z tristimulus spectral curve (sampled at spectral values).

Returns:

[X,Y,Z]: color coordinates X,Y,Z.

Raises:

No exception is raised.

pyradi.rychroma.**loadCIEbar**(*specvec*, *stype*)

Args:

specvec (np.array[N,] or [N,1]): spectral vector in [um] or [cm-1].
stype (str]): type spectral vector wl=wavelength, wn=wavenumber.

Returns:

CIE tristimilus (np.array[:,4]: cols=[specvec,x,y,z])

Raises:

No exception is raised.

pyradi.rychroma.**CIErgbCIExy**(*rgb*)

Convert from CIE RGB coordinates to CIE (x,y) coordinates

The CIE RGB colour space is one of many colour spaces, using three monochromatic primary colours at standardized wavelengths of 700 nm (red), 546.1 nm (green) and 435.8 nm (blue). Other RGB colour spaces uses different primary colours.

This function converts from RGB coordinates (assumed CIE RGB) to CIE xy colour. The rgb array can have any number N of datasets in np.array[N,3]. r, g, and b and in the first, second and third columns.

https://en.wikipedia.org/wiki/CIE_1931_color_space https://en.wikipedia.org/wiki/RGB_color_space

Args:

rgb (np.array[N,3]): CIE red/green/blue colour space component, N sets

Returns:

xy (np.array[N,2]): color coordinates x, y.

Raises:

No exception is raised.

pyradi.rychroma.**CIExyCIErgb**(*xy*)

Convert from CIE RGB coordinates to CIE (x,y) coordinates

The CIE RGB colour space is one of many colour spaces, using three monochromatic primary colours at standardized wavelengths of 700 nm (red), 546.1 nm (green) and 435.8 nm (blue). Other RGB colour spaces uses different primary colours.

This function converts from CIE xy colour to RGB coordinates (assumed CIE RGB). The xy array can have any number N of datasets in np.array[N,2]. x is in the first column and y in the second column

The rgb values are scaled such that the maximum value of any one component is 1, calculated separately per row. In other words, each rgb coordinate is normalised to 255 in one colour.

https://en.wikipedia.org/wiki/CIE_1931_color_space https://en.wikipedia.org/wiki/RGB_color_space

Args:

xy (np.array[N,2]): color coordinates x, y.

Returns:

rgb (np.array[N,3]): CIE red/green/blue colour space component, N sets

Raises:

No exception is raised.

CHAPTER

TWELVE

BULK DETECTOR MODELLING

12.1 Overview

This model was built to give the user a simple but reliable tool to simulate or to understand main parameters used to design a photovoltaic (PV) infrared photodetector. All the work done in this model was based in classical equations found in the literature.

See the __main__ function for examples of use.

The example suggested here uses InSb parameters found in the literature. For every compound or material, all the parameters, as well as the bandgap equation must be changed.

This code uses the scipy.constants physical constants. For more details see http://docs.scipy.org/doc/scipy/reference/constants.html

This code does not yet fully comply with the coding standards

References:

[1] Infrared Detectors and Systems, EL Dereniak & GD Boreman, Wiley [2] Infrared Detectors, A Rogalski (1st or 2nd Edition), CRC Press [3] Band Parameters for III-V Compound Semiconductors and their Alloys, I. Vurgaftmann, J. R. Meyer, and L. R. Ram-Mohan, Journal of Applied Physics 89 11, pp. 5815–5875, 2001.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

12.2 Module functions

pyradi.rydetector.**JouleTeEv**(*EJ*)

Convert energy in Joule to eV.

Args:

EJ: Energy in J

Returns:

EeV: Energy in eV

pyradi.rydetector.**eVtoJoule**(*EeV*)

Convert energy in eV to Joule.

Args:

E: Energy in eV

Returns:

EJ: Energy in J

pyradi.rydetector.**FermiDirac**(*Ef*, *EJ*, *T*)

Returns the Fermi-Dirac probability distribution, given the crystal's Fermi energy, the temperature and the energy where the distribution values is required.

Args:

Ef: Fermi energy in J

EJ: Energy in J

T : Temperature in K

Returns:

fermiD : the Fermi-Dirac distribution

pyradi.rydetector.**Absorption**(*wavelength*, *Eg*, *tempDet*, *a0*, *a0p*)

Calculate the spectral absorption coefficient for a semiconductor material with given material values.

The model used here is based on Equations 3.5, 3.6 in Dereniaks book.

Args:

wavelength: spectral variable [m]

Eg: bandgap energy [Ev]

tempDet: detector's temperature in [K]

a0: absorption coefficient [m-1] (Dereniak Eq 3.5 & 3.6)

a0p: absorption coefficient in [m-1] (Dereniak Eq 3.5 & 3.6)

Returns:

absorption: spectral absorption coefficient in [m-1]

pyradi.rydetector.**AbsorptionFile**(*wavelength*, *filename*)

Read the absorption coefficient from a data file and interpolate on the input spectral range.

The data file must have the wavelength in the first column and absorption coefficient in [m-1] in the second column.

Args:

wavelength: spectral variable [m]

filename: file containing the data

Returns:

wavelength: values where absorption is defined

absorption: spectral absorption coefficient in [m-1]

pyradi.rydetector.**QuantumEfficiency**(*absorption*, *d1*, *d2*, *theta1*, *nFront*, *nMaterial*)

Calculate the spectral quantum efficiency (QE) for a semiconductor material with given absorption and material values.

Args:

absorption: spectral absorption coefficient in [m-1]

d1: depth where the detector depletion layer starts [m]

d2: depth where the detector depletion layer ends [m]

theta1: angle between the surface's normal and the radiation in radians

nFront: index of refraction of the material in front of detector

nMaterial: index of refraction of the detector material

Returns:

quantumEffic: spectral quantum efficiency

pyradi.rydetector.**Responsivity**(*wavelength*, *quantumEffic*)

Responsivity quantifies the amount of output seen per watt of radiant optical power input [1]. But, for

this application it is interesting to define spectral responsivity that is the output per watt of monochromatic radiation.

The model used here is based on Equations 7.114 in Dereniak's book.

Args:

wavelength: spectral variable [m]
quantumEffic: spectral quantum efficiency

Returns:

responsivity in [A/W]

pyradi.rydetector.**DStar**(*areaDet*, *deltaFreq*, *iNoise*, *responsivity*)

The spectral D* is the signal-to-noise output when 1 W of monochromatic radiant flux is incident on 1 cm2 detector area, within a noise-equivalent bandwidth of 1 Hz.

Args:

areaDet: detector's area in [m2]
deltaFreq: measurement or desirable bandwidth - [Hz]
iNoise: noise current [A]
responsivity: spectral responsivity in [A/W]

Returns

detectivity [cm sqrt[Hz] / W] (note units)

pyradi.rydetector.**NEP**(*iNoise*, *responsivity*)

NEP is the radiant power incident on detector that yields SNR=1 [1].

Args:

iNoise: noise current [A]
responsivity: spectral responsivity in [A/W]

Returns

spectral noise equivalent power [W]

pyradi.rydetector.**Isaturation**(*mobE*, *tauE*, *mobH*, *tauH*, *me*, *mh*, *na*, *nd*, *Eg*, *tDetec*, *areaDet*)

This function calculates the reverse saturation current, by Equation 7.22 in Dereniak's book

Args:

mobE: electron mobility [m2/V.s]
tauE: electron lifetime [s]
mobH: hole mobility [m2/V.s]
tauH: hole lifetime [s]
me: electron effective mass [kg]
mh: hole effective mass [kg]
na: acceptor concentration [m-3]
nd: donor concentration [m-3]
Eg: energy bandgap in [Ev]
tDetec: detector's temperature in [K]
areaDet: detector's area [m2]

Returns:

I0: reverse sat current [A]

pyradi.rydetector.**EgVarshni**(*E0*, *VarshniA*, *VarshniB*, *tempDet*)

This function calculates the bandgap at detector temperature, using the Varshni equation

Args:

E0: band gap at room temperature [eV]
VarshniA: Varshni parameter
VarshniB: Varshni parameter
tempDet: detector operating temperature [K]

Returns:

Eg: bandgap at stated temperature [eV]

pyradi.rydetector.**IXV**(*V*, *IVbeta*, *tDetec*, *iPhoto*, *I0*)

This function provides the diode curve for a given photocurrent.

The same function is also used to calculate the dark current, using IVbeta=1 and iPhoto=0

Args:

V: bias [V]
IVbeta: diode equation non linearity factor;
tDetec: detector's temperature [K]
iPhoto: photo-induced current, added to diode curve [A]
I0: reverse sat current [A]

Returns:

current from detector [A]

pyradi.rydetector.**Noise**(*tempDet*, *IVbeta*, *Isat*, *iPhoto*, *vBias=0*)

This function calculates the noise power spectral density produced in the diode: shot noise and thermal noise. The assumption is that all noise sources are white noise PSD.

Eq 5.143 plus thermal noise, see Eq 5.148

Args:

tempDet: detector's temperature [K]
IVbeta: detector nonideal factor [-]
Isat: reverse saturation current [A]
iPhoto: photo current [A]
vBias: bias voltage on the detector [V]

Returns:

detector noise power spectral density [A/Hz1/2]
R0: dynamic resistance at zero bias.
Johnson noise only noise power spectral density [A/Hz1/2]
Shot noise only noise power spectral density [A/Hz1/2]

pyradi.rydetector.**DstarSpectralFlatPhotonLim**(*Tdetec*, *Tenvironment*, *epsilon*)

This function calculates the photon noise limited D* of a detector with unlimited spectral response. This case does not apply to photon detectors. The absorption is assumed spectrally flat.

Args:

Tdetec: detector temperature [K]
Tenvironment: environment temperature [K]
epsilon: emissivity/absorption

Returns:

D* [cm sqrt[Hz] / W] (note units)

CHAPTER

THIRTEEN

STARING ARRAY MODULE (RYSTARE)

13.1 Overview

This module provides a high level model for CCD and CMOS staring array signal chain modelling. The model accepts an input image in photon rate irradiance units and then proceeds to calculate the various noise components and signal components along the signal flow chain.

The code in this module serves as an example of implementation of a high-level CCD/CMOS photosensor signal chain model. The model is described in the article 'High-level numerical simulations of noise in solid-state photosensors: review and tutorial' by Mikhail Konnik and James Welsh, arXiv:1412.4031v1 [astro-ph.IM]. The code was originally written in Matlab and used for the Adaptive Optics simulations and study of noise propagation in wavefront sensors, but can be used for many other applications involving CCD/CMOS photosensors. The original files are available at:

- Paper: http://arxiv.org/pdf/1412.4031.pdf
- Matlab code: https://bitbucket.org/aorta/highlevelsensorsim

The original Matlab code was ported to Python and extended in a number of ways. The core of the model remains the original Konnik model as implemented in the Matlab code. The Python code was validated against results obtained with the Matlab code, up to a point and then substantially reworked and refactored. During the refactoring due diligence was applied with regression testing, checking the new results against the previous results. The results were communicated and confirmed with Konnik. A number of corrections were also made to Konnik's original code, some with his cooperation and some without his involvement.

The documentation in the code was copied from Konnik's Matlab code, so he deserves the credit for the very detailed documentation. His documentation was extracted from the paper quoted above.

The sample Python code (derived from Konnik's code) in the repository models two different cases

- a simple model: which is completely linear (no non-linearities), where all noise are Gaussian, and without source follower noise,
- an advanced model: which has V/V and V/e non-linearities, Wald or lognormal noise, source follower and sense node noise sources and even ADC non-linearities.

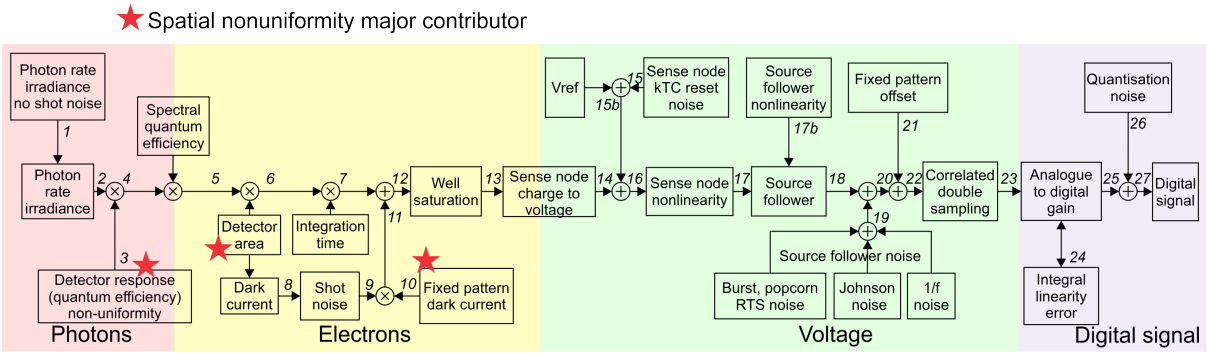
The Python code supports enabling/disabling of key components by using flags.

In the documentation for the Matlab code Konnik expressed the hope "that this model will be useful for somebody, or at least save someone's time. The model can be (and should be) criticized." Indeed it has, thanks Mikhail! Konnik quotes George E. P. Box, the famous statistician, and who said that "essentially, all models are wrong, but some are useful".

13.2 Signal Flow

The process from incident photons to the digital numbers appearing in the image is outlined in the picture below. The input image must be provided in photon rate irradiance units [q/(s.m^2)], with photon noise already present in the image. The count of photons captured in the detector is determined from the irradiance by accounting for the detector area and integration time. Then, the code models the process of conversion from photons to electrons and

subsequently to signal voltage. Various noise sources are modelled to derive at a realistic image model. Finally, the ADC converts the voltage signal into digital numbers. The whole process is depicted in the figure below.



Many noise sources contribute to the resulting noise image that is produced by the sensor. Noise sources can be broadly classified as either *fixed-pattern (time-invariant)* or *temporal (time-variant)* noise. Fixed-pattern noise refers to any spatial pattern that does not change significantly from frame to frame. Temporal noise, on the other hand, changes from one frame to the next. All these noise sources are modelled in the code. For more details see Konnik's original paper or the docstrings present in the code.

13.3 Changes to Matlab code

1. Renamed many, if not all, variables to be more descriptive.
2. Created a number of new functions by splitting up the Matlab functions for increased modularity.
3. Store (almost) all input and output variables in an HDF5 file for full record keeping.
4. Precalculate the image data input as HDF5 files with linear detector parameters embedded in the file. This was done to support future image size calculations. The idea is to embed the target frequency in the data file to relate observed performance with the frequency on the focal plane.
5. Moved sourcefollower calcs out from under dark signal flag. sourcefollower noise is now always calculated irrespective of whether dark noise is selected or not.
6. Input image now photon rate irradiance q/(m2.s), image should already include photon noise in input. Removed from ccd library: irradiance from radiant to photon units, adding photon shot noise. This functionality has been added to the image generation code.
7. Both CCD and CMOS now have fill factors, the user can set CCD fill factor differently from CMOS fill factor. The fill factor value is used as-in in the rest of the code, without checking for CCD or CMOS. This is done because CCD fill factor is 1.0 for full frame sensors but can be less than 1.0 for other types of CCD.
8. Now uses SciPy's CODATA constants where these are available.
9. Put all of the code into a single file rystare.py in the pyradi repository.
10. Minor changes to Konnik's excellent documentation to be Sphinx compatible. Documentation is now generated as part of the pyradi documentation.

13.4 Example Code

The two examples provided by Konnik are merged into a single code, with flags to select between the two options. The code is found at the end of the module file in the *__main__* part of the module file. Set *doTest = 'Simple'* or *doTest = 'Advanced'* depending on which model. Either example will run the *photosensor* function (all functions are thoroughly documented in the Python code, thanks Mikhail!).

The two prepared image files are both 256x256 in size. New images can be generated following the example shown in the *__main__* part of the rystare.py module file (use the function *create_HDF5_image* as a starting point to develop your own).

The easiest way to run the code is to open a command window in the installation directory and run the *run_example* function in the module code. This will load the module and execute the example code function. Running the example code function will create files with names similar to *PSOutput.hdf5* and *PSOutput.txt*. To run the example, create a python file with the following contents and run it at the command line prompt:

```
import pyradi.rystare as rystare
rystare.run_example('Advanced','Output', doPlots=True, doHisto=True, doImages=True)
rystare.run_example('Simple','Output', doPlots=True, doHisto=True, doImages=True)
```

By setting all the flags to True the example code will print a number of images to file. Plotting the results to file takes a while. Execution is much faster with all flags set to False.

Study the text file using a normal text editor and study the HDF5 file by using the viewer available from https://www.hdfgroup.org/products/java/hdfview/.

Some time in future an IPython notebook will be released on https://github.com/NelisW/ComputationalRadiometry.

The full code for the example file is as follows:

```
#prepare so long for Python 3
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import numpy as np
import re
import os.path
from matplotlib import cm as mcm
import matplotlib.mlab as mlab

import pyradi.rystare as rystare
import pyradi.ryplot as ryplot
import pyradi.ryfiles as ryfiles
import pyradi.ryutils as ryutils

"""This file provides examples of use of the CcdCmosSim models for a CMOS/CCD photosensor.

Two models are provided 'simple' and 'advanced'

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf
"""

#set up the parameters for this run
doPlots=True
doHisto=True
doImages=True
outfilename = 'Output'
pathtoimage = 'W:/MyApps/pyradi/pyradi/data/image-Disk-256-256.hdf5'

doTest = 'Simple'
doTest = 'Advanced'

if doTest in ['Simple']:
    prefix = 'PS'
elif  doTest in ['Advanced']:
    prefix = 'PA'
else:
    exit('Undefined test')

[m, cm, mm, mum, nm, rad, mrad] = rystare.define_metrics()
```

```
#open the file to create data structure and store the results, remove if exists
hdffilename = '{}{}.hdf5'.format(prefix, outfilename)
if os.path.isfile(hdffilename):
    os.remove(hdffilename)
strh5 = ryfiles.open_HDF(hdffilename)

#sensor parameters
strh5['rystare/SensorType'] = 'CCD' # must be in capitals
#strh5['rystare/SensorType'] = 'CMOS' # must be in capitals

# full-frame CCD sensors has 100% fil factor (Janesick: 'Scientific Charge-Coupled Devices')
if strh5['rystare/SensorType'].value in ['CMOS']:
    strh5['rystare/FillFactor'] = 0.5 # Pixel Fill Factor for CMOS photo sensors.
else:
    strh5['rystare/FillFactor'] = 1.0 # Pixel Fill Factor for full-frame CCD photo sensors.

strh5['rystare/IntegrationTime'] = 0.01 # Exposure/Integration time, [sec].
strh5['rystare/ExternalQuantumEff'] = 0.8  # external quantum efficiency, fraction not reflected.
strh5['rystare/QuantumYield'] = 1. # number of electrons absorbed per one photon into material bu
strh5['rystare/FullWellElectrons'] = 2e4 # full well of the pixel (how many electrons can be stor
strh5['rystare/SenseResetVref'] = 3.1 # Reference voltage to reset the sense node. [V] typically

#sensor noise
strh5['rystare/SenseNodeGain'] = 5e-6 # Sense node gain, A_SN [V/e]

#source follower
strh5['rystare/SourceFollowerGain'] = 1. # Source follower gain, [V/V], lower means amplify the n

# Correlated Double Sampling (CDS)
strh5['rystare/CDS-Gain'] = 1. # CDS gain, [V/V], lower means amplify the noise.

# Analogue-to-Digital Converter (ADC)
strh5['rystare/ADC-Num-bits'] = 12. # noise is more apparent on high Bits
strh5['rystare/ADC-Offset'] = 0. # Offset of the ADC, in DN

# Light Noise parameters
strh5['rystare/flag/photonshotnoise'] = True #photon shot noise.
# photo response non-uniformity noise (PRNU), or also called light Fixed Pattern Noise (light FPN
strh5['rystare/flag/PRNU'] = True
strh5['rystare/noise/PRNU/seed'] = 362436069
strh5['rystare/noise/PRNU/model'] = 'Janesick-Gaussian'
strh5['rystare/noise/PRNU/parameters'] = [] # see matlab filter or scipy lfilter functions for de
strh5['rystare/noise/PRNU/factor'] = 0.01 # PRNU factor in percent [typically about 1\% for CCD a

# Dark Current Noise parameters
strh5['rystare/flag/darkcurrent'] = True
strh5['rystare/OperatingTemperature'] = 300. # operating temperature, [K]
strh5['rystare/DarkFigureMerit'] = 1. # dark current figure of merit, [nA/cm2].  For very poor se
#  Increasing the DFM more than 10 results to (with the same exposure time of 10^-6):
#  Hence the DFM increases the standard deviation and does not affect the mean value.
strh5['rystare/DarkCurrentElecrons'] = 0.  #to be computed

# dark current shot noise
strh5['rystare/flag/DarkCurrent-DShot'] = True

#dark current Fixed Pattern Noise
strh5['rystare/flag/DarkCurrentDarkFPN-Pixel'] = True
# Janesick's book: dark current FPN quality factor is typically between 10\% and 40\% for CCD and
strh5['rystare/noise/darkFPN/DN'] = 0.3
strh5['rystare/noise/darkFPN/seed'] = 362436128
strh5['rystare/noise/darkFPN/limitnegative'] = True # only used with 'Janesick-Gaussian'

if doTest in ['Simple']:
```

```
    strh5['rystare/noise/darkFPN/model'] = 'Janesick-Gaussian'
    strh5['rystare/noise/darkFPN/parameters'] = [];   # see matlab filter or scipy lfilter functi
elif  doTest in ['Advanced']:
    strh5['rystare/noise/darkFPN/model'] = 'LogNormal' #suitable for long exposures
    strh5['rystare/noise/darkFPN/parameters'] = [0., 0.4] #first is lognorm_mu; second is lognorm_
else:
    pass

# #alternative model
# strh5['rystare/noise/darkFPN/model']  = 'Wald'
# strh5['rystare/noise/darkFPN/parameters']  = 2. #small parameters (w<1) produces extremely narr

# #alternative model
# strh5['rystare/noise/darkFPN/model']  = 'AR-ElGamal'
# strh5['rystare/noise/darkFPN/parameters']  = [1., 0.5] # see matlab filter or scipy lfilter fun

#dark current Offset Fixed Pattern Noise
strh5['rystare/flag/darkcurrent_offsetFPN'] = True
strh5['rystare/noise/darkFPN_offset/model'] = 'Janesick-Gaussian'
strh5['rystare/noise/darkFPN_offset/parameters'] = [] # see matlab filter or scipy lfilter functi
strh5['rystare/noise/darkFPN_offset/DNcolumn'] = 0.0005 # percentage of (V_REF - V_SN)

# Source Follower VV non-linearity
strh5['rystare/flag/VVnonlinearity'] = False

#ADC
strh5['rystare/flag/ADCnonlinearity'] = 0
strh5['rystare/ADC-Gain'] = 0.

if doTest in ['Simple']:
    strh5['rystare/flag/sourcefollowernoise'] = False
elif  doTest in ['Advanced']:
#source follower noise.
    strh5['rystare/flag/sourcefollowernoise'] = True
    strh5['rystare/noise/sf/CDS-SampleToSamplingTime'] = 1e-6 #CDS sample-to-sampling time [sec].
    strh5['rystare/noise/sf/flickerCornerHz'] = 1e6 #flicker noise corner frequency $f_c$ in [Hz]
    strh5['rystare/noise/sf/dataClockSpeed'] = 20e6 #MHz data rate clocking speed.
    strh5['rystare/noise/sf/WhiteNoiseDensity'] = 15e-9 #thermal white noise [\f$V/Hz^{1/2}\f$, t
    strh5['rystare/noise/sf/DeltaIModulation'] = 1e-8 #[A] source follower current modulation ind
    strh5['rystare/noise/sf/FreqSamplingDelta'] = 10000. #sampling spacing for the frequencies (e
else:
    pass

#charge to voltage
strh5['rystare/flag/Venonlinearity'] = False

#sense node reset noise.
strh5['rystare/sn/V-FW'] = 0.
strh5['rystare/sn/V-min'] = 0.
strh5['rystare/sn/C-SN'] = 0.
strh5['rystare/flag/SenseNodeResetNoise'] = True
strh5['rystare/noise/sn/ResetKTC-Sigma'] = 0.
strh5['rystare/noise/sn/ResetFactor'] = 0.8 # the compensation factor of the Sense Node Reset Noi
                                      # 1 - no compensation from CDS for Sense node reset noise.
                                      # 0 - fully compensated SN reset noise by CDS.

#Sensor noises and signal visualisation
strh5['rystare/flag/plots/doPlots'] = False
strh5['rystare/flag/plots/plotLogs'] = False
# strh5['rystare/flag/plots/irradiance'] = True
# strh5['rystare/flag/plots/electrons'] = True
# strh5['rystare/flag/plots/volts'] = True
# strh5['rystare/flag/plots/DN'] = True
```

```
# strh5['rystare/flag/plots/SignalLight'] = True
# strh5['rystare/flag/plots/SignalDark'] = True

#For testing and measurements only:
strh5['rystare/flag/darkframe'] = False # True if no signal, only dark

#============================================================================

if strh5['rystare/flag/darkframe'].value:  # we have zero light illumination
    hdffilename = 'data/image-Zero-256-256.hdf5'
else:   # load an image, nonzero illumination
    hdffilename = 'data/image-Disk-256-256.hdf5'

if pathtoimage is None:
    pathtoimage = os.path.dirname(__file__) + '/' + hdffilename

imghd5 = ryfiles.open_HDF(pathtoimage)

#images must be in photon rate irradiance units q/(m2.s)
strh5['rystare/SignalPhotonRateIrradiance'] = imghd5['image/PhotonRateIrradiance'].value

strh5['rystare/pixelPitch'] = imghd5['image/pixelPitch'].value
strh5['rystare/imageName'] = imghd5['image/imageName'].value
strh5['rystare/imageSizePixels'] = imghd5['image/imageSizePixels'].value

#calculate the noise and final images
strh5 = rystare.photosensor(strh5) # here the Photon-to-electron conversion occurred.

with open('{}{}.txt'.format(prefix,outfilename), 'wt') as fo:
    fo.write('{:25}, {:.5e}, {:.5e}\n'.format('SignalPhotonRateIrradiance',np.mean(strh5['rystare/
    fo.write('{:25}, {:.5e}, {:.5e}\n'.format('signalLight',np.mean(strh5['rystare/signalLight'].v
    fo.write('{:25}, {:.5e}, {:.5e}\n'.format('signalDark',np.mean(strh5['rystare/signalDark'].va
    fo.write('{:25}, {:.5e}, {:.5e}\n'.format('source_follower_noise',np.mean(strh5['rystare/noise
    fo.write('{:25}, {:.5e}, {:.5e}\n'.format('SignalPhotons',np.mean(strh5['rystare/SignalPhotons
    fo.write('{:25}, {:.5e}, {:.5e}\n'.format('SignalElectrons',np.mean(strh5['rystare/SignalElect
    fo.write('{:25}, {:.5e}, {:.5e}\n'.format('SignalVoltage',np.mean(strh5['rystare/SignalVoltage
    fo.write('{:25}, {:.5e}, {:.5e}\n'.format('SignalDN',np.mean(strh5['rystare/SignalDN'].value),

if doPlots:
    lstimgs = ['rystare/SignalPhotonRateIrradiance','rystare/SignalPhotons','rystare/SignalElectro
               'rystare/SignalDN','rystare/signalLight','rystare/signalDark', 'rystare/noise/PRNU/
               'rystare/noise/darkFPN/nonuniformity']
    # ryfiles.plotHDF5Images(strh5, prefix=prefix, colormap=mcm.gray,  lstimgs=lstimgs, logscale=
    ryfiles.plotHDF5Images(strh5, prefix=prefix, colormap=mcm.jet,  lstimgs=lstimgs, logscale=str

if doHisto:
    lstimgs = ['rystare/SignalPhotonRateIrradiance','rystare/SignalPhotons','rystare/SignalElectro
               'rystare/SignalDN','rystare/signalLight','rystare/signalDark',
               'rystare/noise/PRNU/nonuniformity','rystare/noise/darkFPN/nonuniformity']
    ryfiles.plotHDF5Histograms(strh5, prefix, bins=100, lstimgs=lstimgs)

if doImages:
    lstimgs = ['rystare/SignalPhotonRateIrradiance','rystare/SignalPhotonRate', 'rystare/SignalPho
               'rystare/SignalDN','rystare/signalLight','rystare/signalDark', 'rystare/noise/sn_
               'rystare/noise/PRNU/nonuniformity','rystare/noise/darkFPN/nonuniformity']
    ryfiles.plotHDF5Bitmaps(strh5, prefix, format='png', lstimgs=lstimgs)

strh5.flush()
strh5.close()
```

13.5 HDF5 File

The Python implementation of the model uses an HDF5 file to capture the input and output data for record keeping or subsequent analysis. HDF5 files provide for hierarchical data structures and easy read/save to disk. See the file *hdf5-as-data-format.md* (*[hdf5asdataformat]*) in the pyradi root directory for more detail.

Input images are written to and read from HDF5 files as well. These files store the image as well as the images' dimensional scaling in the focal plane. The intent is to later create test targets with specific spatial frequencies in these files.

13.6 Code Overview

This module provides a high level model for CCD and CMOS staring array signal chain modelling. The work is based on a paper and Matlab code by Mikhail Konnik, available at:

- Paper available at: http://arxiv.org/pdf/1412.4031.pdf
- Matlab code available at: https://bitbucket.org/aorta/highlevelsensorsim

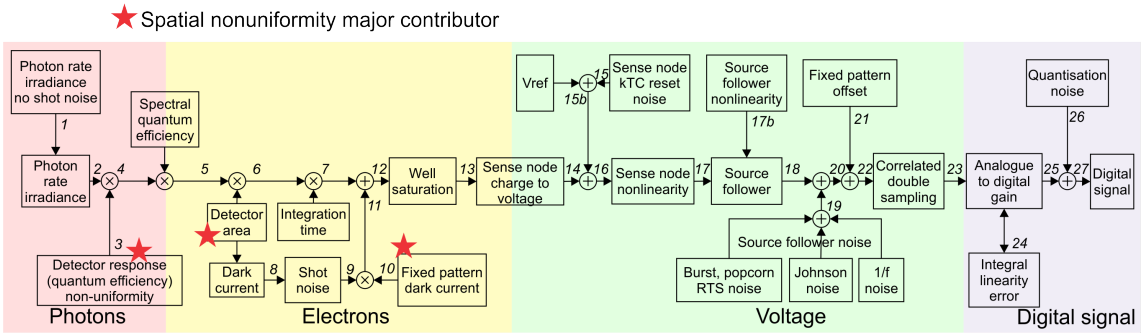
See the documentation at http://nelisw.github.io/pyradi-docs/_build/html/index.html or pyradi/doc/rystare.rst for more detail.

13.7 Module functions

`pyradi.rystare.`**`photosensor`**(*strh5*)

This routine simulates the behaviour of a CCD/CMOS sensor, performing the conversion from irradiance to electrons, then volts, and then digital numbers.

The process from incident photons to the digital numbers appeared on the image is outlined. First of all, the radiometry is considered. Then, the process of conversion from photons to electrons is outlined. Following that, conversion from electrons to voltage is described. Finally, the ADC converts the voltage signal into digital numbers. The whole process is depicted on Figure below.



Many noise sources contribute to the resulting noise image that is produced by photosensors. Noise sources can be broadly classified as either *fixed-pattern (time-invariant)* or *temporal (time-variant)* noise. Fixed-pattern noise refers to any spatial pattern that does not change significantly from frame to frame. Temporal noise, on the other hand, changes from one frame to the next.

Note that in the sequence below we add signal and noise signals linearly together. For uncorrelated noise sources, the noise power values are added in quadrature, but that does not apply here, because we are adding instantaneous noise values (per pixel) so that these noise and signal values add linearly.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**set_photosensor_constants**(*strh5*)

Defining the constants that are necessary for calculation of photon energy, dark current rate, etc.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**check_create_datasets**(*strh5*)

Create the arrays to store the various image-sized variables.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

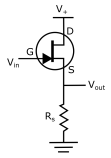
Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**source_follower**(*strh5*)

The amplification of the voltage from Sense Node by Source Follower.

Conventional sensor use a floating-diffusion sense node followed by a charge-to-voltage amplifier, such as a source follower.



Source follower is one of basic single-stage field effect transistor (FET) amplifier topologies that is typically used as a voltage buffer. In such a circuit, the gate terminal of the transistor serves as the input, the source is the output, and the drain is common to both input and output. At low frequencies, the source follower has voltage gain:

$A_{\mathrm{v}} = \frac{v_{\mathrm{out}}}{v_{\mathrm{in}}} = \frac{g_m R_{\mathrm{S}}}{g_m R_{\mathrm{S}}+1} \approx 1 \qquad (g_m R_{\mathrm{S}} \gg 1)$

Source follower is a voltage follower, its gain is less than 1. Source followers are used to preserve the linear relationship between incident light, generated photoelectrons and the output voltage.

The V/V non-linearity affect shot noise (but does not affect FPN curve) and can cause some shot-noise probability density compression. The V/V non-linearity non-linearity is caused by non-linear response in ADC or source follower.

The V/V non-linearity can be simulated as a change in source follower gain A_{SF} as a linear function of signal:

$A_{SF_{new}} = \alpha \cdot \frac{V_{REF} - S(V_{SF})}{V_{REF}} + A_{SF}$,

where $\alpha = A_{SF} \cdot \frac{\gamma_{nlr}-1}{V_{FW}}$ and γ_{nlr} is a non-linearity ratio of A_{SF}. In the simulation we assume $A_{SF} = 1$ and $\gamma_{nlr} = 1.05$ i.e. 5% of non-linearity of A_{SF}. Then the voltage is multiplied on the new sense node gain $A_{SF_{new}}$:

$I_V = I_V \cdot A_{SF_{new}}$

After that, the voltage goes to ADC for quantisation to digital numbers.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**fixed_pattern_offset**(*strh5*)

Add dark fixed pattens offset

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**cds**(*strh5*)

Reducing the noise by Correlated Double Sampling, but right now the routine just adds the noise.

Correlated Double Sampling (CDS) is a technique for measuring photo voltage values that removes an undesired noise. The sensor's output is measured twice. Correlated Double Sampling is used for compensation of Fixed pattern noise caused by dark current leakage, irregular pixel converters and the like. It appears on the same pixels at different times when images are taken. It can be suppressed with noise reduction and on-chip noise reduction technology. The main approach is CDS, having one light signal read by two circuits.

In CDS, a circuit measures the difference between the reset voltage and the signal voltage for each pixel, and assigns the resulting value of charge to the pixel. The additional step of measuring the output node reference voltage before each pixel charge is transferred makes it unnecessary to reset to the same level for each pixel.

First, only the noise is read. Next, it is read in combination with the light signal. When the noise component is subtracted from the combined signal, the fixed-pattern noise can be eliminated.

CDS is commonly used in image sensors to reduce FPN and reset noise. CDS only reduces offset FPN (gain FPN cannot be reduced using CDS). CDS in CCDs, PPS, and photogate APS, CDS reduces reset noise, in photodiode APS it increases it See Janesick's book and especially El Gamal's lectures.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

`pyradi.rystare.adc` (*strh5*)

An analogue-to-digital converter (ADC) transforms a voltage signal into discrete codes.

An analogue-to-digital converter (ADC) transforms a voltage signal into discrete codes. An N-bit ADC has 2^N possible output codes with the difference between code being $V_{ADC.REF}/2^N$. The resolution of the ADC indicates the number of discrete values that can be produced over the range of analogue values and can be expressed as:

$K_{ADC} = \frac{V_{ADC.REF} - V_{\min}}{N_{max}}$ where $V_{\text{ADC.REF}}$ is the maximum voltage that can be quantified, V_{min} is minimum quantifiable voltage, and $N_{max} = 2^N$ is the number of voltage intervals. Therefore, the output of an ADC can be represented as:

$ADC_{Code} = \text{round}\left(\frac{V_{input} - V_{min}}{K_{ADC}}\right)$

The lower the reference voltage $V_{ADC.REF}$, the smaller the range of the voltages one can measure.

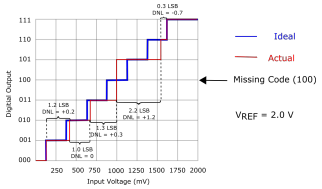
After the electron matrix has been converted to voltages, the sense node reset noise and offset FPN noise are added, the V/V gain non-linearity is applied (if desired), the ADC non-linearity is applied (if necessary). Finally the result is multiplied by ADC gain and rounded to produce the signal as a digital number:

$I_{DN} = \text{round}(A_{ADC} \cdot I_{total.V})$,

where $I_{\text{total.V}} = (V_{ADC.REF} - I_V)$ is the total voltage signal accumulated during one frame acquisition, $V_{ADC.REF}$ is the maximum voltage that can be quantified by an ADC, and I_V is the total voltage signal accumulated by the end of the exposure (integration) time and conversion. Usually $I_V = I_{SN.V}$ after the optional V/V non-linearity is applied. In this case, the conversion from voltages to digital signal is linear. The adcnonlinearity "non-linear ADC case is considered below".

In terms of the ADC, the following non-linearity and noise should be considered for the simulations of the photosensors: Integral Linearity Error, Differential Linearity Error, quantisation error, and ADC offset.

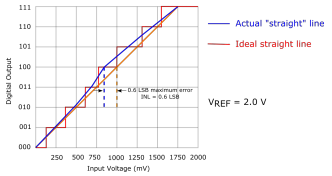
The DLE indicates the deviation from the ideal 1 LSB (Least Significant Bit) step size of the analogue input signal corresponding to a code-to-code increment. Assume that the voltage that corresponds to a step of 1 LSB is V_{LSB}. In the ideal case, a change in the input voltage of V_{LSB} causes a change in the digital code of 1 LSB. If an input voltage that is more than V_{LSB} is required to change a digital code by 1 LSB, then the ADC has DLE error. In this case, the digital output remains constant when the input voltage changes from, for example, $2V_{LSB}$ to $4V_{LSB}$, therefore corresponding the digital code can never appear at the output. That is, that code is missing.



In the illustration above, each input step should be precisely 1/8 of reference voltage. The first code transition from 000 to 001 is caused by an input change of 1 LSB as it should be. The second transition, from

001 to 010, has an input change that is 1.2 LSB, so is too large by 0.2 LSB. The input change for the third transition is exactly the right size. The digital output remains constant when the input voltage changes from 4 LSB to 5 LSB, therefore the code 101 can never appear at the output.

The ILE is the maximum deviation of the input/output characteristic from a straight line passed through its end points. For each voltage in the ADC input, there is a corresponding code at the ADC output. If an ADC transfer function is ideal, the steps are perfectly superimposed on a line. However, most real ADC's exhibit deviation from the straight line, which can be expressed in percentage of the reference voltage or in LSBs. Therefore, ILE is a measure of the straightness of the transfer function and can be greater than the differential non-linearity. Taking the ILE into account is important because it cannot be calibrated out.



For each voltage in the ADC input there is a corresponding word at the ADC output. If an ADC is ideal, the steps are perfectly superimposed on a line. But most of real ADC exhibit deviation from the straight line, which can be expressed in percentage of the reference voltage or in LSBs.

In our model, we simulate the Integral Linearity Error (ILE) of the ADC as a dependency of ADC gain $A_{ADC.linear}$ on the signal value. Denote $\gamma_{ADC.nonlin}$ as an ADC non-linearity ratio (e.g., $\gamma_{ADC.nonlin} = 1.04$). The linear ADC gain can be calculated from Eq.~ref{eq:kadc} as $A_{ADC} = 1/K_{ADC}$ and used as $A_{ADC.linear}$. The non-linearity coefficient α_{ADC} is calculated as:

$$\alpha_{ADC} = \frac{1}{V_{ADC.REF}} \left(\frac{\log(\gamma_{ADC.nonlin} \cdot A_{ADC.linear})}{\log(A_{ADC.linear})} - 1 \right)$$

where $V_{\mathrm{ADC.REF}}$ is the maximum voltage that can be quantified by an ADC:

$$A_{ADC.nonlin} = A_{ADC.linear}^{1-\alpha_{ADC} I_{total.V}},$$

where $A_{ADC.linear}$ is the linear ADC gain. The new non-linear ADC conversion gain $A_{ADC.nonlin}$ is then used for the simulations.

Quantisation errors are caused by the rounding, since an ADC has a finite precision. The probability distribution of quantisation noise is generally assumed to be uniform. Hence we use the uniform distribution to model the rounding errors.

It is assumed that the quantisation error is uniformly distributed between -0.5 and +0.5 of the LSB and uncorrelated with the signal. Denote q_{ADC} the quantising step of the ADC. For the ideal DC, the quantisation noise is:

$$\sigma_{ADC} = \sqrt{\frac{q_{ADC}^2}{12}}.$$

If $q_{ADC} = 1$ then the quantisation noise is $\sigma_{ADC} = 0.29$ DN. The quantisation error has a uniform distribution. We do not assume any particular architecture of the ADC in our high-level sensor model. This routine performs analogue-to-digital convertation of volts to DN.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

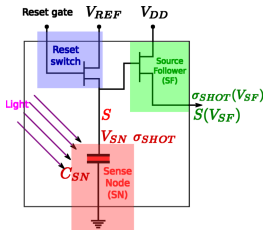
`pyradi.rystare.`**`charge_to_voltage`**(*strh5*)

The charge to voltage conversion occurs inside this routine

V/e nonlinearity is small for CCD detetors, but can be very high for some CMOS architectures (up to 200%) [from Janesick p87]

A new matrix strh5['rystare/signal/voltage'] is created and the raw voltage signal is stored.

After the charge is generated in the pixel by photo-effect, it is moved row-by-row to the sense amplifier that is separated from the pixels in case of CCD. The packets of charge are being shifted to the output sense node, where electrons are converted to voltage. The typical sense node region is presented on Figure below.



Sense node is the final collecting point at the end of the horizontal register of the CCD sensor. The CCD pixels are made with MOS devices used as reverse biased capacitors. The charge is readout by a MOSFET based charge to voltage amplifier. The output voltage is inversely proportional to the sense node capacitor. Typical example is that the sense node capacitor of the order $50fF$, which produces a gain of $3.2\mu V/e^-$. It is also important to minimize the noise of the output amplifier, textbf{typically the largest noise source in the system}. Sense node converts charge to voltage with typical sensitivities $1\ldots4\mu V/e^-$.

The charge collected in each pixel of a sensor array is converted to voltage by sense capacitor and source-follower amplifier.

Reset noise is induced during such conversion. Prior to the measurement of each pixel's charge, the CCD sense capacitor is reset to a reference level. Sense node converts charge to voltage with typical sensitivities $1\ldots4\mu V/e^-$. The charge collected in each pixel of a sensor array is converted to voltage by sense capacitor and source-follower amplifier. Reset noise is induced during such conversion. Prior to the measurement of each pixel's charge, the CCD sense node capacitor is reset to a reference level.

Sense Node gain non-linearity, or V/e non-linearity

The V/e^- non-linearity affect both FPN and shot noise and can cause some shot-noise probability density compression. This type of non-linearity is due to sense node gain non-linearity. Then sense node sensitivity became non-linear (see Janesick's book):

$$S_{SN}(V_{SN}/e^-) = \frac{S(V_{SN})}{(k_1/q)\ln(V_{REF}/[V_{REF}-S(V_{SN})])}$$

The V/e^- non-linearity can be expressed as a non-linear dependency of signals in electron and a sense-node voltage:

$$S[e^-] = \frac{k1}{q}\ln\left[\frac{V_{REF}}{V_{REF}-S(V_{SN})}\right]$$

The V/e^- non-linearity affects photon shot noise and skews the distribution, however this is a minor effect. The V/e^- non-linearity can also be thought as a sense node capacitor non-linearity: when a small signal is measured, C_{SN} is fixed or changes negligible; on the other hand, C_SN changes significantly and that can affect the signal being measured.

For the simulation purpose, the V/e^- non-linearity can be expressed as:

$$V_{SN} = V_{REF} - S(V_{SN}) = V_{REF}\exp\left[-\frac{\alpha\cdot S[e^-]\cdot q}{k1}\right]$$

where $k1 = 10.909 * 10^{-15}$ and q is the charge of an electron, and α is the coefficient of non-linearity strength. The capacitance is given by C = k1/V

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**sense_node_reset_noise**(*strh5*)

This routine calculates the noise standard deviation for the sense node reset noise.

Sense node Reset noise (kTC noise)

Prior to the measurement of each pixel's charge packet, the sense node capacitor is reset to a reference voltage level. Noise is generated at the sense node by an uncertainty in the reference voltage level due to thermal variations in the channel resistance of the MOSFET reset transistor. The reference level of the sense capacitor is therefore different from pixel to pixel.

Because reset noise can be significant (about 50 rms electrons), most high-performance photosensors incorporate a noise-reduction mechanism such as correlated double sampling (CDS).

kTC noise occurs in CMOS sensors, while for CCD sensors the sense node reset noise is removed~ (see Janesick's book) by Correlated Double Sampling (CDS). Random fluctuations of charge on the sense node during the reset stage result in a corresponding photodiode reset voltage fluctuation. The sense node reset noise (in volt units) is given by:

$\sigma_{RESET} = \sqrt{\frac{k_B T}{C_{SN}}}$

By the relationship Q=CV it can be shown that the kTC noise can be expressed as electron count by

$\sigma_{RESET} = \sqrt{\frac{k_B T C_{SN}}{q}}$

see also https://en.wikipedia.org/wiki/Johnson%E2%80%93Nyquist_noise

The simulation of the sense node reset noise may be performed as an addition of non-symmetric probability distribution to the reference voltage V_{REF}. However, the form of distribution depends on the sensor's architecture and the reset technique. An Inverse-Gaussian distribution can be used for the simulation of kTC noise that corresponds to a hard reset technique in the CMOS sensor, and the Log-Normal distribution can be used for soft-reset technique. The sense node reset noise can be simulated for each (i, j)-th pixel for the soft-reset case as:

$I_{SN.reset.V} = ln\mathcal{N}(0, \sigma^2_{RESET})$

then added to the matrix $I_{REF.V}$ in Volts that corresponds to the reference voltage.

Note: For CCD, the sense node reset noise is entirely removed by CDS.

Note: In CMOS photosensors, it is difficult to remove the reset noise for the specific CMOS pixels architectures even after application of CDS. Specifically, the difficulties arise in 'rolling shutter' and 'snap' readout modes. The reset noise is increasing after CDS by a factor of $\sqrt{2}$. Elimination of reset noise in CMOS is quite challenging.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**dark_current_and_dark_noises**(*strh5*)

This routine for adding dark current signals and noise, including dark FPN and dark shot noise.

This model is taken from Janesick's 'Photon Transfer' book, page 168, which in turn is taken from Janesick's 'Scientific Charge-Coupled Devices' book, page 622.

The dark signal is calculated for all pixels in the model. It is implemented using *ones* function in MATLAB as a matrix of the same size as the simulated photosensor. For each (i, j)-th pixel we have:

$I_{dc.e^-} = t_I \cdot D_R$,

where D_R is the average dark current (originally derived for silicon):

$D_R = 2.55 \cdot 10^{15} P_A D_{FM} T^{1.5} \exp\left[-\frac{E_{gap}}{2 \cdot k \cdot T}\right]$,

where: D_R is in units of [e^{-1}/s], P_A is the pixel's area [cm^2]; D_{FM} is the dark current figure-of-merit in units of [nA/cm^2] at 300K, varies significantly with detector material and sensor manufacturer, and used in this simulations as 0.5 nA/cm^2 for silicon; E_{gap} is the bandgap energy of the semiconductor which also varies with temperature; k is Boltzman's constant that is $8.617 \cdot 10^{-5}[eV/K]$.

The relationship between band gap energy and temperature can be described by Varshni's empirical expression,

$E_{gap}(T) = E_{gap}(0) - \frac{\alpha T^2}{T+\beta}$,

where $E_{gap}(0)$, α and β are material constants. The energy bandgap of semiconductors tends to decrease as the temperature is increased. This behaviour can be better understood if one considers that the inter-atomic spacing increases when the amplitude of the atomic vibrations increases due to the increased thermal energy. This effect is quantified by the linear expansion coefficient of a material.

For the Silicon: $E_{gap}(0) = 1.1557[eV]$**,** $\alpha = 7.021 * 10^{-4}$ **[eV/K], and** $\beta = 1108$ [K].

It appears that fill factor does not apply to dark noise (Janesick book p168 and Konnik's code does not show this).

According to Janesick's Photon transfer book p169 the dark current FPN standard deviation is around 10% (CCD) and 40% (CMOS) of the dark current. Note that 'dark' FPN (DN) is much greater than 'light' FPN (PN) by approximately 10 to 40 times.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields, current in nA

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**source_follower_noise**(*strh5*)

The source follower noise routine, calculates noise in volts.

The pixel's source follower noise limits the read noise, however in high-end CCD and CMOS cameras the source follower noise has been driven down to one electron rms. Pixel source follower MOSFET noise consists of three types of noise: - white noise; - flicker noise; - random telegraph noise (RTS). Each type of noise has its own physics that will be briefly sketched below.

Johnson noise (white noise)

Similarly to the reset noise in sense node, the source-follower amplifier MOSFET has a resistance that generates thermal noise whose value is governed by the Johnson white noise equation. It is therefore either referred to as Johnson noise or simply as white noise, since its magnitude is independent of frequency. If

the effective resistance is considered to be the output impedance of the source-follower amplifier, the white noise, in volts, is determined by the following equation:

$N_{white}(V_{SF}) = \sqrt{4kTBR_{SF}}$

where k is Boltzmann's constant (J/K), T is temperature [K], B refers to the noise power bandwidth [Hz], and R_{SF} is the output impedance of the source-follower amplifier.

Flicker noise

The flicker noise is commonly referred to as $1/f$ noise because of its approximate inverse dependence on frequency. For cameras in which pixels are read out at less than approximately 1 megahertz, and with a characteristic $1/f$ noise spectrum, the read noise floor is usually determined by 1/f noise. Note that the noise continues to decrease at this rate until it levels off, at a frequency referred to as the $1/f$ corner frequency. For the typical MOSFET amplifier, the white noise floor occurs at approximately 4.5 $nV/Hz^{1/2}$.

Prominent sources of $1/f$ noise in an image sensor are pink-coloured noise generated in the photo-diodes and the low-bandwidth analogue operation of MOS transistors due to imperfect contacts between two materials. Flicker noise is generally accepted to originate due to the existence of interface states in the image sensor silicon that turn on and off randomly according to different time constants. All systems exhibiting 1/f behaviour have a similar collection of randomly-switching states. In the MOSFET, the states are traps at the silicon-oxide interface, which arise because of disruptions in the silicon lattice at the surface. The level of $1/f$ noise in a CCD sensor depends on the pixel sampling rate and from certain crystallographic orientations of silicon wafer.

Random Telegraph Signal (RTS) noise

As the CCD and CMOS pixels are shrinking in dimensions, the low-frequency noise increases. In such devices, the low-frequency noise performance is dominated by Random Telegraph Signals (RTS) on top of the 1/f noise. The origin of such an RTS is attributed to the random trapping and de-trapping of mobile charge carriers in traps located in the oxide or at the interface. The RTS is observed in MOSFETs as a fluctuation in the drain current. A pure two-level RTS is represented in the frequency domain by a Lorentzian spectrum.

Mathematically the source follower's noise power spectrum can be described as: $S_{SF}(f) = W(f)^2 \cdot \left(1 + \frac{f_c}{f}\right) + S_{RTS}(f)$,

where $W(f)$ is the thermal white noise [$V/Hz^{1/2}$, typically $15nV/Hz^{1/2}$], flicker noise corner frequency f_c in [Hz] (flicker noise corner frequency is the frequency where power spectrum of white and flicker noise are equal), and the RTS power spectrum is given (see Janesick's book):

$S_{RTS}(f) = \frac{2\Delta I^2 \tau_{RTS}}{4+(2\pi f \tau_{RTS})^2}$,

where τ_{RTS} is the RTS characteristic time constant [sec] and ΔI is the source follower current modulation induced by RTS [A].

The source follower noise can be approximated as:

$$\sigma_{SF} = \frac{\sqrt{\int_0^\infty S_{SF}(f) H_{CDS}(f) df}}{A_{SN} A_{SF} (1 - \exp^{-t_s/\tau_D})}$$

where: - σ_{SF} is the source follower noise [e- rms] - f is the electrical frequency [Hz] - t_s is the CDS sample-to-sampling time [sec] - τ_D is the CDS dominant time constant (see Janesick's Scientific CCDs book) usually set as $\tau_D = 0.5t_s$ [sec].

The $H_{CDS}(f)$ function is the CDS transfer function is (see Janesick's book):

$H_{CDS}(f) = \frac{1}{1+(2\pi f \tau_D)^2} \cdot [2 - 2\cos(2\pi f t_s)]$

First term sets the CDS bandwidth for the white noise rejection before sampling takes place through $B = 1/(4\tau_D)$, where B is defined as the noise equivalent bandwidth [Hz].

Note: In CCD photosensors, source follower noise is typically limited by the flicker noise.

Note: In CMOS photosensors, source follower noise is typically limited by the RTS noise. As a side note, such subtle kind of noises is visible only on high-end ADC like 16 bit and more.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**multiply_detector_area**(*strh5*)

This routine multiplies detector area

The input to the model of the photosensor is assumed to be a matrix $E_q \in R^{N \times M}$ that has been converted to electronrate irradiance, corresponding to electron rate [e/(m2.s)]. The electron rate irriance is converted to electron rate into the pixel by accounting for detector area:

$\Phi_q = \text{round}\left(E_q \cdot P_A\right),$

where P_A is the area of a pixel [m2].

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**multiply_integration_time**(*strh5*)

This routine multiplies with integration time

The input to the model of the photosensor is assumed to be a matrix $E_q \in R^{N \times M}$ that has been converted to electrons, corresponding to electron rate [e/s]. The electron rate is converted to electron count into the pixel by accounting for detector integration time:

$\Phi_q = \text{round}\left(E_q \cdot t_I\right),$

where t_I is integration (exposure) time.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**convert_to_electrons**(*strh5*)

This routine converts photon rate irradiance to electron rate irradiance

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**shotnoise**(*sensor_signal_in*)

This routine adds photon shot noise to the signal of the photosensor that is in photons.

The photon shot noise is due to the random arrival of photons and can be described by a Poisson process. Therefore, for each (i, j)-th element of the matrix Φ_q that contains the number of collected photons, a photon shot noise is simulated as a Poisson process $\mathcal{P}$ with mean Λ:

$\Phi_{ph.shot} = \mathcal{P}(\Lambda), \quad \text{where } \Lambda = \Phi_q.$

We use the *ryutils.poissonarray* function that generates Poisson random numbers with mean Λ. That is, the number of collected photons in (i, j)-th pixel of the simulated photosensor in the matrix Φ_q is used as the mean Λ for the generation of Poisson random numbers to simulate the photon shot noise. The input of the *ryutils.poissonarray* function will be the matrix Φ_q that contains the number of collected photons. The output will be the matrix $\Phi_{ph.shot} \rightarrow \Phi_q$, i.e., the signal with added photon shot noise. The matrix $\Phi_{ph.shot}$ is recalculated each time the simulations are started, which corresponds to the temporal nature of the photon shot noise.

Args:

sensor_signal_in (np.array[N,M]): photon irradiance in, in photons

Returns:

sensor_signal_out (np.array[N,M]): photon signal out, in photons

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**responsivity_FPN_light**(*strh5*)

Multiploying the photon signal with the PRNU.

The Photo Response Non-Uniformity (PRNU) is the spatial variation in pixel conversion gain (from photons to electrons). When viewing a uniform scene the pixel signals will differ because of the PRNU, mainly due to variations in the individual pixel's characteristics such as detector area and spectral response. These variations occur during the manufacture of the substrate and the detector device.

The PRNU is signal-dependent (proportional to the input signal) and is fixed-pattern (time-invariant). For visual (silicon) sensors the PRNU factor is typically $0.01 \ldots 0.05$, but for HgCdTe sensors it can be as large as $0.02 \ldots 0.25$. It varies from sensor to sensor, even within the same manufacturing batch.

The photo response non-uniformity (PRNU) is considered as a temporally-fixed light signal non-uniformity. The PRNU is modelled using a Gaussian distribution for each (i, j)-th pixel of the matrix I_{e^-}, as $I_{PRNU.e^-} = I_{e^-}(1 + \mathcal{N}(0, \sigma^2_{PRNU}))$ where σ_{PRNU} is the PRNU factor value.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**responsivity_FPN_dark**(*strh5*)

Add dark current noises that consist of Dark FPN and Dark shot noise.

Pixels in a hardware photosensor cannot be manufactured exactly the same from perfectly pure materials. There will always be variations in the photo detector area that are spatially uncorrelated, surface defects at the SiO_2/Si interface (see Sakaguchi paper on dark current reduction), and discrete randomly-distributed charge generation centres. These defects provide a mechanism for thermally-excited carriers to move between the valence and conduction bands. Consequently, the average dark signal is not uniform but has a spatially-random and fixed-pattern noise (FPN) structure. The dark current FPN can be expressed as follows:

$\sigma_{d.FPN} = t_I D_R \cdot D_N,$

where t_I is the integration time, D_R is the average dark current, and D_N is the dark current FPN factor that is typically $0.1 \ldots 0.4$ for CCD and CMOS sensors.

There are also so called 'outliers' or 'dark spikes'; that is, some pixels generate a dark signal values much higher than the mean value of the dark signal. The mechanism of such 'dark spikes' or 'outliers' can be described by the Poole-Frenkel effect (increase in emission rate from a defect in the presence of an electric field).

Simulation of dark current fixed pattern noise

The dark current Fixed Pattern Noise (FPN) is simulated using non-symmetric distributions to account for the 'outliers' or 'hot pixels'. It is usually assumed that the dark current FPN can be described by Gaussian distribution. However, such an assumption provides a poor approximation of a complicated noise picture.

Studies show that a more adequate model of dark current FPN is to use non-symmetric probability distributions. The concept is to use two distributions to describe very 'leaky' pixels that exhibit higher noise level than others. The first distribution is used for the main body of the dark current FPN, with a uniform distribution superimposed to model 'leaky' pixels. For simulations at room-temperature (25° C) authors use a logistic distribution, where a higher proportion of the population is distributed in the tails. For higher temperatures, inverse Gaussian and Log-Normal distributions have been proposed. The Log-Normal distribution works well for conventional 3T APS CMOS sensors with comparatively high dark current.

In our simulations we use the Log-Normal distribution for the simulation of dark current FPN in the case of short integration times, and superimposing other distributions for long integration times. The actual simulation code implements various models, including Log-Normal, Gaussian, and Wald distribution to emulate the dark current FPN noise for short- and long-term integration times.

The dark current FPN for each pixel of the matrix $I_{dc.shot.e^-}$ is computed as:

$I_{dc.FPN.e^-} = I_{dc.shot.e^-} + I_{dc.shot.e^-} \cdot ln\mathcal{N}(0, \sigma^2_{dc.FPN.e^-})$

where $\sigma_{dc.FPN.e^-} = t_I D_R D_N$, D_R is the average dark current, and D_N is the dark current FPN factor. Since the dark current FPN does not change from one frame to the next, the matrix $ln\mathcal{N}$ is calculated once and then can be re-used similar to the PRNU simulations.

The experimental results confirm that non-symmetric models, and in particular the Log-Normal distribution, adequately describe the dark current FPN in CMOS sensors, especially in the case of a long integration time (longer than 30-60 seconds). For long-exposure case, one needs to superimpose two (or more, depending on the sensor) probability distributions.

Args:

strh5 (hdf5 file): hdf5 file that defines all simulation parameters

Returns:

in strh5: (hdf5 file) updated data fields

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**FPN_models**(*sensor_signal_rows*, *sensor_signal_columns*, *noisetype*, *noisedistribution*, *spread*, *filter_params=None*)

The routine contains various models on simulation of Fixed Pattern Noise.

There are many models for simulation of the FPN: some of the models are suitable for short-exposure time modelling (Gaussian), while other models are more suitable for log-exposure modelling of dark current FPN.

Gaussian model (Janesick-Gaussian)

Fixed-pattern noise (FPN) arises from changes in dark currents due to variations in pixel geometry during fabrication of the sensor. FPN increases exponentially with temperature and can be measured in dark conditions. Column FPN is caused by offset in the integrating amplifier, size variations in the integrating capacitor CF, channel charge injection from reset circuit. FPN components that are reduced by CDS. Dark current FPN can be expressed as:

$\sigma_{D_{FPN}} = D \cdot D_N$,

where D_N is the dark current FPN quality, which is typically between 10% and 40% for CCD and CMOS sensors (see Janesick's book), and $D = t_I D_R$. There are other models of dark FPN, for instance as a autoregressive process.

El Gamal model of FPN with Autoregressive process

To capture the structure of FPN in a CMOS sensor we express $F_{i,j}$ as the sum of a column FPN component Y_j and a pixel FPN component $X_{i,j}$. Thus, $F_{i,j} = Y_j + X_{i,j}$, where the Y_j's and the $X_{i,j}$'s are zero mean random variables.

The first assumption is that the random processes Y_j and $X_{i,j}$ are uncorrelated. This assumption is reasonable since the column and pixel FPN are caused by different device parameter variations. We further assume that the column (and pixel) FPN processes are isotropic.

The idea to use autoregressive processes to model FPN was proposed because their parameters can be easily and efficiently estimated from data. The simplest model, namely first order isotropic autoregressive processes is considered. This model can be extended to higher order models, however, the results suggest that additional model complexity may not be warranted.

The model assumes that the column FPN process Y_j is a first order isotropic autoregressive process of the form:

$Y_j = a(Y_{j-1} + Y_{j+1}) + U_j$

where the U_j s are zero mean, uncorrelated random variables with the same variance σ_U , and $0 \leq a \leq 1$ is a parameter that characterises the dependency of Y_j on its two neighbours.

The model assumes that the pixel FPN process $X_{i,j}$ is a two dimensional first order isotropic autoregressive process of the form:

$X_{i,j} = b(X_{i-1,j} + X_{i+1,j} + X_{i,j-1} + X_{i,j+1}) + V_{i,j}$

where the $V_{i,j}$ s are zero mean uncorrelated random variables with the same variance σ_V , and $0 \leq b \leq 1$ is a parameter that characterises the dependency of $X_{i,j}$ on its four neighbours.

Args:

sensor_signal_rows(int): number of rows in the signal matrix

sensor_signal_columns(int): number of columns in the signal matrix

noisetype(string): type of noise to generate: ['pixel' or 'column']

noisedistribution(string): the probability distribution name ['AR-ElGamal', 'Janesick-Gaussian', 'Wald', 'LogNormal']

spread(float): spread around mean value (sigma/chi/lambda) for the probability distribution
filter_params([nd.array]): a vector of parameters for the probability filter

Returns:

noiseout (np.array[N,M]): generated noise of FPN.

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**nEcntLLightDF**(*tauAtmo*, *tauFilt*, *tauOpt*, *quantEff*, *rhoTarg*, *cosTarg*, *inttime*, *pfrac*, *detarea*, *fno*, *scenario*, *specBand*, *dfPhotRates*)

Calculate the number of electrons in a detector given sensor parameters and photon radiance dataframe

All values in base SI units

Args:

tauAtmo (scalar or nd.array): atmosphere transmittance
tauFilt (scalar or nd.array): sensor filter transmittance
tauOpt (scalar or nd.array): sensor optics transmittance
quantEff (scalar or nd.array): sensor detector quantum efficiency
rhoTarg (scalar or nd.array): target diffuse reflectance
cosTarg (scalar): cos of illuminator angle wrt normal vector
inttime (scalar): integration time s
pfrac (scalar): fraction of clear optics
detarea (scalar): detector area m2
fno (scalar): f number
scenario (str): Scenario as key to rypflux.py dataframe
specBand (str): Spectral band as key to rypflux.py dataframe
dfPhotRadiance (pd.DataFrame): rypflux.py dataframe radiance in q/(s.m2.sr)

Returns:

n (float): number of electrons in charge well

Raises:

No exception is raised.

Author: CJ Willers

pyradi.rystare.**nEcntLLightPhotL**(*tauAtmo*, *tauFilt*, *tauOpt*, *quantEff*, *rhoTarg*, *cosTarg*, *inttime*, *pfrac*, *detarea*, *fno*, *photRadiance*)

Calculate the number of electrons in a detector given sensor parameters and photon radiance

All values in base SI units

Args:

tauAtmo (scalar or nd.array): atmosphere transmittance
tauFilt (scalar or nd.array): sensor filter transmittance
tauOpt (scalar or nd.array): sensor optics transmittance
quantEff (scalar or nd.array): sensor detector quantum efficiency
rhoTarg (scalar or nd.array): target diffuse reflectance
cosTarg (scalar): cos of illuminator angle wrt normal vector
inttime (scalar): integration time s
pfrac (scalar): fraction of clear optics
detarea (scalar): detector area m2
fno (scalar): f number

photRadiance (scalar): in-band photon radiance q/(s.m2.sr)

Returns:

n (float): number of electrons in charge well

Raises:

No exception is raised.

Author: CJ Willers

pyradi.rystare.**nElecCntThermalScene**(*wl*, *tmptr*, *emis*, *tauAtmo*, *tauFilt*, *tauOpt*, *quantEff*, *inttime*, *pfrac*, *detarea*, *fno*)

Calculate the number of electrons in a detector from a thermal source

All values in base SI units

Args:

wl (np.array): wavelength vector
tmptr (scalar): source temperature
emis (np.array of scalar): source emissivity
tauAtmo (scalar or nd.array): atmosphere transmittance
tauFilt (scalar or nd.array): sensor filter transmittance
tauOpt (scalar or nd.array): sensor optics transmittance
quantEff (scalar or nd.array): sensor detector quantum efficiency
inttime (scalar): integration time s
pfrac (scalar): fraction of clear optics
detarea (scalar): detector area m2
fno (scalar): f number

Returns:

n (float): number of electrons in charge well

Raises:

No exception is raised.

Author: CJ Willers

pyradi.rystare.**nEcntThermalOptics**(*wl*, *tmptrOpt*, *tauFilt*, *tauOpt*, *quantEff*, *inttime*, *pfrac*, *detarea*, *fno*)

Calculate the number of electrons in a detector from hot optics

All values in base SI units

Args:

wl (np.array): wavelength vector
tmptrOpt (scalar): optics temperature
tauFilt (scalar or nd.array): sensor filter transmittance
tauOpt (scalar or nd.array): sensor optics transmittance
quantEff (scalar or nd.array): sensor detector quantum efficiency
inttime (scalar): integration time s
pfrac (scalar): fraction of clear optics
detarea (scalar): detector area m2
fno (scalar): f number

Returns:

n (float): number of electrons in charge well

Raises:

No exception is raised.

Author: CJ Willers

pyradi.rystare.**nElecCntReflSun**(*wl*, *tauSun*, *tauAtmo=1*, *tauFilt=1*, *tauOpt=1*, *quantEff=1*, *rhoTarg=1*, *cosTarg=1*, *inttime=1*, *pfrac=1*, *detarea=1*, *fno=0.8862269255*, *emissun=1.0*, *tmprt=6000.0*)

Calculate the number of electrons in a detector or photon radiance for reflected sunlight

All values in base SI units.

By using the default values when calling the function the radiance at the source can be calculated.

Args:

wl (np.array (N,) or (N,1)): wavelength
tauSun (np.array (N,) or (N,1)): transmittance between the scene and sun
tauAtmo (np.array (N,) or (N,1)): transmittance between the scene and sensor
tauFilt (np.array (N,) or (N,1)): sensor filter transmittance
tauOpt (np.array (N,) or (N,1)): sensor optics transmittance
quantEff (np.array (N,) or (N,1)): detector quantum efficiency
rhoTarg (np.array (N,) or (N,1)): target diffuse surface reflectance
cosTarg (scalar): cosine between surface normal and sun/moon direction
inttime (scalar): detector integration time
pfrac (scalar): fraction of optics clear aperture
detarea (scalar): detector area
fno (scalar): optics fnumber
emissun (scalar): sun surface emissivity
tmprt (scalar): sun surface temperature

Returns:

n (scalar): number of electrons accumulated during integration time

Raises:

No exception is raised.

pyradi.rystare.**darkcurrentnoise**(*inttime*, *detarea*, *temptr*, *Egap*, *DFM=5e-06*)

Calculate the dark current noise given detector parameters

Args:

inttime (scalar): integration time in seconds
detarea (scalar): detector area in m2
temptr (scalar): temperature in K
Egap (scalar): bandgap in eV
DFM (scalar): in units of nA/m2

Returns:

n (scalar): dark current noise as number of electrons

Raises:

No exception is raised.

pyradi.rystare.**kTCnoiseCsn**(*temptr*, *sensecapacity*)

Args:

temptr (scalar): temperature in K
sensecapacity (): sense node capacitance F

Returns:

n (scalar): noise as number of electrons

Raises:

No exception is raised.

pyradi.rystare.**kTCnoiseGv**(*temptr*, *gv*)

Args:

temptr (scalar): temperature in K

gv (scalar): sense node gain V/e

Returns:

n (scalar): noise as number of electrons

Raises:

No exception is raised.

pyradi.rystare.**create_HDF5_image**(*imageName*, *imtype*, *pixelPitch*, *numPixels*, *fracdiameter=0*, *fracblurr=0*, *irrad_scale=1*, *irrad_min=0*, *wavelength=5.5e-07*, *steps=10*)

This routine performs makes a simple illuminated circle with blurred boundaries.

Then the sensor's radiant irradiance in units [W/m2] are converted to photon rate irradiance in units [q/m2.s)] by relating one photon's energy to power at the stated wavelength by $Q_p = \frac{h \cdot c}{\lambda}$, where λ is wavelength, h is Planck's constant and c is the speed of light.

The image file is in HDF5 format, containing the input parameters to the image creation process. A few minimum entries are required, but you can add any information youo wish to document the data. The following minimum HDF5 entries are required by pyradi.rystare:

`'image/imageName'` (string): the image name

`'image/PhotonRateIrradianceNoNoise'` np.array[M,N]: a float array with the image pixel values no noise

`'image/PhotonRateIrradiance'` np.array[M,N]: a float array with the image pixel values with noise

`'image/pixelPitch'`: ([float, float]): detector pitch in m [row,col]

`'image/imageSizePixels'`: ([int, int]): number of pixels [row,col]

`'image/imageFilename'` (string): the image file name

`'image/wavelength'` (float): where photon rate calcs are done um

`'image/imageSizeRows'` (int): the number of image rows

`'image/imageSizeCols'` (int): the number of image cols

`'image/imageSizeDiagonal'` (float): the FPA diagonal size in mm

`'image/irradianceLux'` (float): the maximum luminous exitance in the image lux (optional)

`'image/irradianceWatts'` (float): the maximum exitance in the image W/m2 (optional)

`'image/temperature'` (float): the maximum target temperature in the image K (optional)

Args:

imageName (string): the image name, used to form the filename

imtype (string): string to define the type if image to be created ['zeros','disk','stairslin','stairslog']

pixelPitch ([float, float]): detector pitch in m [row,col]

numPixels ([int, int]): number of pixels [row,col]

fracdiameter (float): diameter of the disk as fraction of minimum image size

fracblurr (float): blurr of the disk as fraction of minimum image size

irrad_scale (float): multiplicative scale factor (max value)

irrad_min (float): additive minimum value in the image

wavelength (float): wavelength where photon rate calcs are done in [m]

steps (int): number of steps in the stairs image

Returns:

nothing: as a side effect an image file is written

Raises:

No exception is raised.

Author: CJ Willers

pyradi.rystare.**define_metrics**()

This simple routine defines various handy shorthand for cm and mm in the code.

The code defines a number of scaling factors to convert to metres and radians

Args:

None

Returns:

scaling factors.

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**limitzero**(*a*, *thr=0.6*)

Performs an asymetric clipping to prevent negative values. The lower-end values are clumped up towards the lower positive values, while upper-end values are not affected.

This function is used to prevent negative random variables for wide sigma and low mean value, e.g., N(1,.5). If the random variables are passed through this function The resulting distribution is not normal any more, and has no known analytical form.

A threshold value of around 0.6 was found to work well for N(1,small) up to N(1,.5).

Before you use this function, first check the results using the code below in the main body of this file.

Args:

a (np.array): an array of floats,

Returns:

scaling factors.

Raises:

No exception is raised.

Author: CJ Willers

pyradi.rystare.**run_example**(*doTest=u'Advanced'*, *outfilename=u'Output'*, *pathtoimage=None*, *doPlots=False*, *doHisto=False*, *doImages=False*)

This code provides examples of use of the pyradi.rystare model for a CMOS/CCD photosensor.

Two models are provided 'simple' and 'advanced'

doTest can be 'Simple' or 'Advanced'

Args:

doTest (string): which example to run 'Simple', or 'Advanced'
outfilename (string): filename for output files
pathtoimage (string): fully qualified path to where the image is located
doPlots (boolean): flag to control the creation of false colour image plots with colour bars
doHisto (boolean): flag to control the creation of image histogram plots
doImages (boolean): flag to control the creation of monochrome image plots

Returns:

hdffilename (string): output HDF filename

Raises:

No exception is raised.

Author: Mikhail V. Konnik, revised/ported by CJ Willers

Original source: http://arxiv.org/pdf/1412.4031.pdf

pyradi.rystare.**get_summary_stats**(*hdffilename*)

Return a string with all the summary input and results data.

Args:

hdffilename (string): filename for input HDF file

Returns:

Returns a string with summmary data.

Raises:

No exception is raised.

Author: CJ Willers

CHAPTER

FOURTEEN

PROBABILITY TOOLS (RYPROB)

14.1 Overview

The functions in this module is used in the staring array model. It was originally used in Konnik's staring array model (he sourced it from somewhere else). Credits for the original source are included in each of the functions.

14.2 Code Overview

This module provides a high level model for CCD and CMOS staring array signal chain modelling. The work is based on a paper and Matlab code by Mikhail Konnik, available at:

- Paper available at: http://arxiv.org/pdf/1412.4031.pdf
- Matlab code available at: https://bitbucket.org/aorta/highlevelsensorsim

See the documentation at http://nelisw.github.io/pyradi-docs/_build/html/index.html or pyradi/doc/rystare.rst for more detail.

14.3 Module functions

pyradi.ryprob.**distribution_exp**(*distribParams*, *out*, *funcName*)

Exponential Distribution

This function is meant to be called via the *distributions_generator* function.

pdf $= \lambda * \exp(-\lambda * y)$

cdf $= 1 - \exp(-\lambda * y)$

•Mean = 1/lambda

•Variance = 1/lambda^2

•Mode = lambda

•Median = log(2)/lambda

•Skewness = 2

•Kurtosis = 6

GENERATING FUNCTION: $T = -\log_e(U)/\lambda$

PARAMETERS: distribParams[0] is lambda - inverse scale or rate (lambda>0)

SUPPORT: y, y>= 0

CLASS: Continuous skewed distributions

NOTES: The discrete version of the Exponential distribution is the Geometric distribution.

USAGE:

- y = randraw('exp', lambda, sampleSize) - generate sampleSize number of variates from the Exponential distribution with parameter 'lambda';

EXAMPLES:

1. y = randraw('exp', 1, [1 1e5]);
2. y = randraw('exp', 1.5, 1, 1e5);
3. y = randraw('exp', 2, 1e5);
4. y = randraw('exp', 3, [1e5 1]);

SEE ALSO: GEOMETRIC, GAMMA, POISSON, WEIBULL distributions http://en.wikipedia.org/wiki/Exponential_distribution

pyradi.ryprob.**distribution_lognormal**(*distribParams*, *out*, *funcName*)

THe Log-normal Distribution (sometimes: Cobb-Douglas or antilognormal distribution)

This function is meant to be called via the *distributions_generator* function.

pdf = 1/(y*sigma*sqrt(2*pi)) * exp(-1/2*((log(y)-mu)/sigma)^2) cdf = 1/2*(1 + erf((log(y)-mu)/(sigma*sqrt(2))));

- Mean = exp(mu + sigma^2/2);
- Variance = exp(2*mu+sigma^2)*(exp(sigma^2)-1);
- Skewness = (exp(1)+2)*sqrt(exp(1)-1), for mu=0 and sigma=1;
- Kurtosis = exp(4) + 2*exp(3) + 3*exp(2) - 6; for mu=0 and sigma=1;
- Mode = exp(mu-sigma^2);

PARAMETERS: mu - location, sigma - scale (sigma>0)

SUPPORT: y, y>0

CLASS: Continuous skewed distribution

NOTES:

1. The LogNormal distribution is always right-skewed
2. Parameters mu and sigma are the mean and standard deviation of y in (natural) log space.
3. mu = log(mean(y)) - 1/2*log(1 + var(y)/(mean(y))^2)
4. sigma = sqrt(log(1 + var(y)/(mean(y))^2))

USAGE:

- randraw('lognorm', [], sampleSize) - generate sampleSize number of variates from the standard Lognormal distribution with location parameter mu=0 and scale parameter sigma=1
- randraw('lognorm', [mu, sigma], sampleSize) - generate sampleSize number of variates from the Lognormal distribution with location parameter 'mu' and scale parameter 'sigma'

EXAMPLES:

1. y = randraw('lognorm', [], [1 1e5]);
2. y = randraw('lognorm', [0, 4], 1, 1e5);
3. y = randraw('lognorm', [-1, 10.2], 1e5);
4. y = randraw('lognorm', [3.2, 0.3], [1e5 1]);

pyradi.ryprob.**distribution_inversegauss**(*distribParams*, *out*, *funcName*)

The Inverse Gaussian Distribution

This function is meant to be called via the *distributions_generator* function.

The Inverse Gaussian distribution is left skewed distribution whose location is set by the mean with the profile determined by the scale factor. The random variable can take a value between zero and infinity. The skewness increases rapidly with decreasing values of the scale parameter.

pdf(y) = sqrt(_lambda/(2*pi*y^3)) * exp(-_lambda./(2*y).*(y/mu-1).^2)

cdf(y) = normcdf(sqrt(_lambda./y).*(y/mu-1)) + exp(2*_lambda/mu)*normcdf(sqrt(_lambda./y).*(-y/mu-1))

where normcdf(x) = 0.5*(1+erf(y/sqrt(2))); is the standard normal CDF

- •Mean = mu
- •Variance = mu^3/_lambda
- •Skewness = sqrt(9*mu/_lambda)
- •Kurtosis = 15*mean/scale
- •Mode = mu/(2*_lambda)*(sqrt(9*mu^2+4*_lambda^2)-3*mu)

PARAMETERS: mu - location; (mu>0), _lambda - scale; (_lambda>0)

SUPPORT: y, y>0

CLASS: Continuous skewed distribution

NOTES:

1. There are several alternate forms for the PDF, some of which have more than two parameters
2. The Inverse Gaussian distribution is often called the Inverse Normal
3. Wald distribution is a special case of The Inverse Gaussian distribution where the mean is a constant with the value one.
4. The Inverse Gaussian distribution is a special case of The Generalized Hyperbolic Distribution

USAGE:

- •randraw('ig', [mu, _lambda], sampleSize) - generate sampleSize number of variates from the Inverse Gaussian distribution with parameters mu and _lambda;

EXAMPLES:

1. y = randraw('ig', [0.1, 1], [1 1e5]);
2. y = randraw('ig', [3.2, 10], 1, 1e5);
3. y = randraw('ig', [100.2, 6], 1e5);
4. y = randraw('ig', [10, 10.5], [1e5 1]);

SEE ALSO: WALD distribution

Method:

There is an efficient procedure that utilizes a transformation yielding two roots. If Y is Inverse Gauss random variable, then following [1] we can write: V = _lambda*(Y-mu)^2/(Y*mu^2) ~ Chi-Square(1)

i.e. V is distributed as a _lambda-square random variable with one degree of freedom. So it can be simply generated by taking a square of a standard normal random number. Solving this equation for Y yields two roots:

y1 = mu + 0.5*mu/_lambda * (mu*V - sqrt(4*mu*_lambda*V + mu^2*V.^2)); and y2 = mu^2/y1;

In [2] showed that Y can be simulated by choosing y1 with probability mu/(mu+y1) and y2 with probability 1-mu/(mu+y1)

References: [1] Shuster, J. (1968). On the Inverse Gaussian Distribution Function, Journal of the American Statistical Association 63: 1514-1516.

[2] Michael, J.R., Schucany, W.R. and Haas, R.W. (1976). Generating Random Variates Using Transformations with Multiple Roots, The American Statistician 30: 88-90.

http://en.wikipedia.org/wiki/Inverse_Gaussian_distribution

pyradi.ryprob.**distribution_logistic**(*distribParams*, *out*, *funcName*)

The Logistic Distribution

This function is meant to be called via the *distributions_generator* function.

The logistic distribution is a symmetrical bell shaped distribution. One of its applications is an alternative to the Normal distribution when a higher proportion of the population being modeled is distributed in the tails.

pdf(y) = exp((y-a)/k)./(k*(1+exp((y-a)/k)).^2)

cdf(y) = 1 ./ (1+exp(-(y-a)/k))

- Mean = a
- Variance = k^2*pi^2/3
- Skewness = 0
- Kurtosis = 1.2

PARAMETERS: a - location, k - scale (k>0);

SUPPORT: y, -Inf < y < Inf

CLASS: Continuous symmetric distribution

USAGE:

- randraw('logistic', [], sampleSize) - generate sampleSize number of variates from the standard Logistic distribution with location parameter a=0 and scale parameter k=1;
- Logistic distribution with location parameter 'a' and scale parameter 'k';

EXAMPLES:

1.y = randraw('logistic', [], [1 1e5]);

2.y = randraw('logistic', [0, 4], 1, 1e5);

3.y = randraw('logistic', [-1, 10.2], 1e5);

4.y = randraw('logistic', [3.2, 0.3], [1e5 1]);

Method:

Inverse CDF transformation method.

http://en.wikipedia.org/wiki/Logistic_distribution

pyradi.ryprob.**distribution_wald**(*distribParams*, *out*, *funcName*)

The Wald Distribution

This function is meant to be called via the *distributions_generator* function.

The Wald distribution is as special case of the Inverse Gaussian Distribution where the mean is a constant with the value one.

pdf = sqrt(chi/(2*pi*y^3)) * exp(-chi./(2*y).*(y-1).^2);

- Mean = 1
- Variance = 1/chi
- Skewness = sqrt(9/chi)
- Kurtosis = 3+ 15/scale

PARAMETERS: chi - scale parameter; (chi>0)

SUPPORT: y, y>0

CLASS: Continuous skewed distributions

USAGE:

•randraw('wald', chi, sampleSize) - generate sampleSize number of variates from the Wald distribution with scale parameter 'chi';

EXAMPLES:

1.y = randraw('wald', 0.5, [1 1e5]);

2.y = randraw('wald', 1, 1, 1e5);

3.y = randraw('wald', 1.5, 1e5);

4.y = randraw('wald', 2, [1e5 1]);

pyradi.ryprob.**distributions_generator**(*distribName=None*, *distribParams=None*, *sampleSize=None*)

The routine contains various models for simulation of FPN (DSNU or PRNU).

This function allows the user to select the distribution by name and pass requisite parameters in a list (which differs for different distrubutions). The size of the distribution is defined by a scalar or list.

sampleSize follows Matlab conventions:

•if None then return a single scalar value

•if scalar int N then return NxN array

•if tuple then return tuple-sized array

Possible values for distribName:

'exp','exponential'
'lognorm','lognormal','cobbdouglas','antilognormal'
'ig', 'inversegauss', 'invgauss'
'logistic'
'wald'

Args:

distribName (string): required distribution name
distribParams ([float]): list of distribution parameters (see below)
sampleSize (None,int,[int,int]): Size of the returned random set

Returns:

out (float, np.array[N,M]): set of random variables for selected distribution.

Raises:

No exception is raised.

The routine set generates various types of random distributions, and is based on the code randraw by Alex Bar Guy & Alexander Podgaetsky These programs are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Author: Alex Bar Guy, comments to alex@wavion.co.il

pyradi.ryprob.**validateParam**(*funcName=None*, *distribName=None*, *runDistribName=None*, *distribParamsName=None*, *paramName=None*, *param=None*, *conditionStr=None*)

Validate the range and number of parameters

Args:

funcName (string): distribution name
distribName (string): distribution name
runDistribName (string): run distribution name

distribParamsName
paramName
param
conditionStr

Returns:

True if the requirements are matched

Raises:

No exception is raised.

pyradi.ryprob.**checkParamsNum**(*funcName*, *distribName*, *runDistribName*, *distribParams*, *correctNum*)

See if the correct number of parameters was supplied. More than one number may apply

Args:

funcName (string): distribution name
distribName (string): distribution name
distribParams ([float]): list of distribution parameters (see below)
correctNum ([int]): list with the possible numbers of parameters

Returns:

True if the requirements are matched

Raises:

No exception is raised.

CHAPTER

FIFTEEN

OUTDOOR SCENE FLUX LEVELS (RYPFLUX)

The module provides photonrate flux levels for outdoor scenes.

15.1 Overview

Provides a simple, order of magnitude estimate of the photon flux and electron count in a detector for various sources and scene lighting. All models are based on published information or derived herein, so you can check their relevancy and suitability for your work.

For a detailed theoretical derivation and more examples of use see: http://nbviewer.jupyter.org/github/NelisW/ComputationalRadiometry/blob/master/07-Optical-Sources.ipynb

See the __main__ function for examples of use.

This package was partly developed to provide additional material in support of students and readers of the book Electro-Optical System Analysis and Design: A Radiometry Perspective, Cornelius J. Willers, ISBN 9780819495693, SPIE Monograph Volume PM236, SPIE Press, 2013. http://spie.org/x648.html?product_id=2021423&origin_id=x646

15.2 Module classes

class `pyradi.rypflux.`**`PFlux`**

See here: https://github.com/NelisW/ComputationalRadiometry/blob/master/07-Optical-Sources.ipynb for mathemetical derivations and more detail.

`lllPhotonrates`(*specranges=None*)

Calculate the approximate photon rate radiance for low light conditions

The colour temperature of various sources are used to predict the photon flux. The calculation uses the colour temperature of the source and the ratio of real low light luminance to the luminance of a Planck radiator at the same temperature as the source colour temperature.

This procedure critically depends on the sources' spectral radiance in the various different spectral bands. For this calculation the approach is taken that for natural scenes the spectral shape can be modelled by a Planck curve at the appropriate colour temperature.

The steps followed are as follows:

1. Calculate the photon rate for the scene at the appropriate colour temperature, spectrally weighted by the eye's luminous efficiency response. Do this for photopic and scotopic vision.
2. Weigh the photopic and scotopic photon rates according to illumination level
3. Determine the ratio k of low light level scene illumination to photon irradiance. This factor k is calculated in the visual band, but then applied to scale the other spectral bands by the same scale.

4.Use Planck radiation at the appropriate colour temperature to calculate the radiance in any spectral band, but then scale the value with the factor k.

The specranges format is a dictionary where the key is the spectral band, and the entry against each key is a list containing two items: the spectral vector and the associated spectral band definition. The third entry in the list must be 'wn' (=wavenumber) or 'wl' (=wavelength) to signify the type of spectral variable One simple example definition is as follows:

```
numpts = 300
specranges = {
    key: [wavelength vector, response vector ],
    'VIS': [np.linspace(0.43,0.69,numpts).reshape(-1,1),np.ones((numpts,1)), 'wl' ],
    'NIR': [np.linspace(0.7, 0.9,numpts).reshape(-1,1),np.ones((numpts,1)), 'wl' ],
    'SWIR': [np.linspace(1.0, 1.7,numpts).reshape(-1,1),np.ones((numpts,1)), 'wl' ],
    'MWIR': [np.linspace(3.6,4.9,numpts).reshape(-1,1),np.ones((numpts,1)), 'wl' ],
    'LWIR': [np.linspace(7.5,10,numpts).reshape(-1,1),np.ones((numpts,1)), 'wl' ],
    }
```

If specranges is None, the predefined values are used, as shown above.

The function returns scene radiance in a Pandas datatable with the following columns containing the spectrally weighted integrated radiance:

```
u'Irradiance-lm/m2', u'ColourTemp', u'FracPhotop', u'k',
u'Radiance-q/(s.m2.sr)-NIR', u'Radiance-q/(s.m2.sr)-VIS',
u'Radiance-q/(s.m2.sr)-MWIR', u'Radiance-q/(s.m2.sr)-LWIR',
u'Radiance-q/(s.m2.sr)-SWIR'
```

and rows with the following index:

```
u'Overcast night', u'Star light', u'Quarter moon', u'Full moon',
u'Deep twilight', u'Twilight', u'Very dark day', u'Overcast day',
u'Full sky light', u'Sun light'
```

Args:

specranges (dictionary): User-supplied dictionary defining the spectral responses. See the dictionary format above and an example in the code.

Returns:

Pandas dataframe with radiance in the specified spectral bands.
The dataframe contains integrated and spectral radiance.

Raises:

No exception is raised.

CHAPTER

SIXTEEN

CODING GUIDELINES

Broadly speaking we adhere to the Google Python Style Guide, but not always. The style guide is available at http://google-styleguide.googlecode.com/svn/trunk/pyguide.html. This style is based on Python's PEP 8 http://www.python.org/dev/peps/pep-0008/.

16.1 Naming Rules

We deviate from PEP 8 / Google's naming rules as shown here. Essentially we avoid underscores inside names, and prefer to Capitalise words to highlight. The primary motivation is (in our opinion) improved readability: it better binds the words into a single entity. Underscores tend to break the name visually into separate sub-names.

Type	Public	Internal	PEP 8
Packages	lowerwordslater		lower_with_under
Modules	lowerwordslater	_lowerwordslater	lower_with_under
Classes	CapWordsLater	_CapWordsLater	CapWords
Exceptions	CapWordsLater		CapWords
Functions	lowerWords-Later()	_lowerWordsLater()	lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER	CAPS_WITH_UNDER
Global/Class Variables	lowerWords-Later	_lowerWordsLater	lower_with_under
Instance Variables	lowerWords-Later	_lowerWordsLater (protected) or __lowerWordsLater (private)	lower_with_under
Method Names	lowerWords-Later()	_lowerWordsLater() (protected) or __lowerWordsLater() (private)	lower_with_under()
Function/Method Parameters	lowerWords-Later		lower_with_under
Local Variables	lowerWords-Later		lower_with_under

CHAPTER

SEVENTEEN

EXAMPLES OF CODE USE

The respective python files all have examples of use and simple test code at the end of each of the files. Execute these python files as scripts to execute the example code, and observe the results.

CHAPTER

EIGHTEEN

INDICES AND TABLES

- genindex
- modindex
- search

[SPIE8543Pyradi] *Pyradi: an open-source toolkit for infrared calculation and data processing*, SPIE Proceedings Vol 8543, Security+Defence 2011, Technologies for Optical Countermeasures, Edinburgh, 24-27 September, C.J. Willers, M. S. Willers, R.A.T. Santos, P.J. van der Merwe, J.J. Calitz, A de Waal and A.E. Mudau.

[hdf5asdataformat] https://github.com/NelisW/pyradi/blob/master/pyradi/hdf5-as-data-format.md

p