pyradi Documentation

Release

pyradi team

April 28, 2014

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)

http://code.google.com/p/pyradi (latest version in the repository)

See docs at

http://pyradi.googlecode.com/svn//trunk/doc/_build/html/index.html

Visit the google group at

http://groups.google.com/group/pyradi-dev

IPython notebooks demonstrating the use of pyradi is available at

https://github.com/NelisW/ComputationalRadiometry#computational-optical-radiometry-with-pyradi

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.
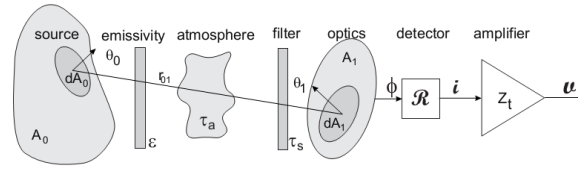

The amplifier output signal can be calculated in the following equation, by integrating over all wavelengths, over the full source area A_0 and over the optical aperture area A_1,

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

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),

$$\int_0^\infty \epsilon_\lambda L_\lambda(T) \tau_{a\lambda} \tau_{s\lambda} \mathcal{R}_\lambda \, d\lambda,$$
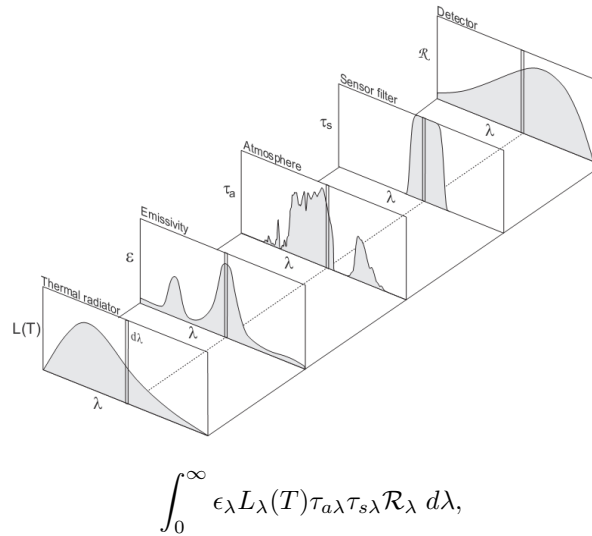
where the spectral variability of the source, medium and sensor parameters are multiplied as spectral variables and afterwards integrated over all wavelengths to yield the total in-band signal. The domain of spectral quantities can be stated in terms of a wavelength, wavenumber, or less often, temporal 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.
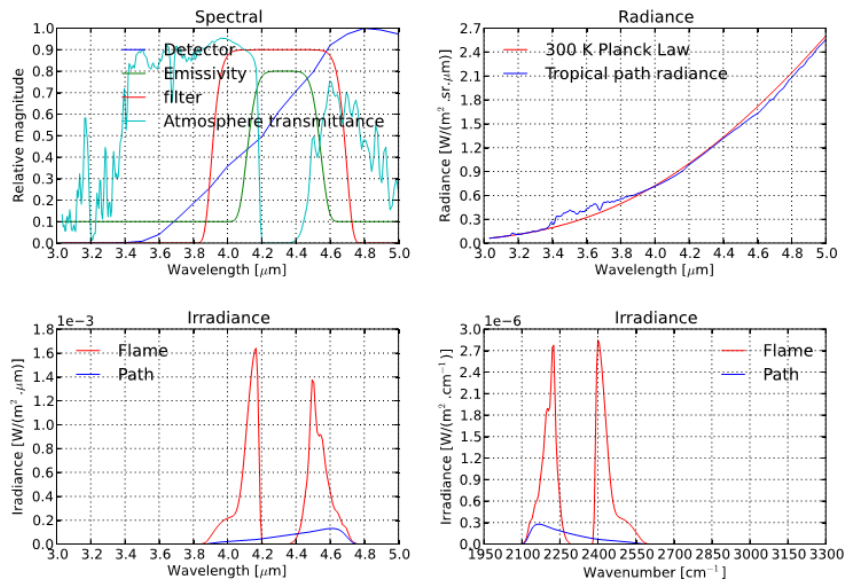
The pyradi code to model this sensor is available as exflamesensor.py. The output from this script is as follows:

```
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.

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.

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=u' %:/,.\\[]'*)
> Clean a string by removing selected characters.

Creates a legal and 'clean' sourcestring from a string by removing some clutter and illegals. 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 source.

Returns:

> (string): A cleaned-up string.

Raises:

> No exception is raised.

---

pyradi.ryfiles.**listFiles**(*root*, *patterns=u'*'*, *recurse=1*, *return_folders=0*, *useRegex=False*)
Lists the files/directories meeting specific requirement

Searches 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.

Args:

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

patterns (string): glob pattern for filename matching

recurse (unt): should the search extend to subdirs of root?

return_folders (int): should foldernames also be returned?

useRegex (bool): should regular expression evaluation be used?

Returns:

A list with matching file/directory names

Raises:

No exception is raised.

pyradi.ryfiles.**readRawFrames**(*fname*, *rows*, *cols*, *vartype*, *loadFrames=[ ]*)
Constructs a numpy array from data in a binary file with known data-type.

Args:

fname (string): path and filename

rows (int): number of rows in frames

cols (int): number of columns in frames

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.**arrayToLaTex**(*filename*, *arr*, *header=None*, *leftCol=None*, *formatstring=u'%1.4e'*, *filemode=u'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): output path and filename

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

header (string): column header in final latex format

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

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

---

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

Returns:

None, writes a file to disk

Raises:

No exception is raised.

`pyradi.ryfiles.`**`epsLaTexFigure`**(*filename*, *epsname*, *caption*, *scale*, *filemode=u'a'*)

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

Args:

fname (string): output path and filename

epsname (string): filename/path to eps file

caption (string): figure caption

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

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

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 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.

> **Args:**

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

> **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.

> **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 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*)
> Untar a tar archive, and save all files to 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:**

>> tgzFilename (string): the name of the tar archive.
>> url (string): url where to look for the file.

---

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

tarFilename (string): unzipped tar filename

Returns:

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

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

Raises:

Exceptions are handled internally and signaled by return value.

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.

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)*)
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.

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*, *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*)
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)

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)

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, cmap=<matplotlib.colors.LinearSegmentedColormap object at 0x05105E90>, cbarshow=False, cbarorientation='vertical', cbarcustomticks=[], cbarfontsize=12, drawGrid=True, xInvert=False, yInvert=False, zInvert=False, logScale=False, elevation=30, azimuth=45, alpha=1*)*
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 with no 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, otherwise fill mesh with colour map (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)

elevation (float): view elevation in degrees (optional)

azimuth (float): view azimuth in degrees (optional)

alpha (float): surface and mesh transparency (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 0x05105E90>, cbarshow=False, cbarorientation='vertical', cbarcustomticks=[], cbarfontsize=12, drawGrid=False, yInvert=False, xInvert=False, contourFill=True, contourLine=True, logScale=False, negativeSolid=False, zeroContourLine=False, contLabel=False, contFmt='%.2f', contCol='k', contFonSz=8, contLinWid=0.5*)

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): type of shading, 'flat' | 'gouraud' (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 (bool): draw the contour at zero (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)

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, pltaxis=None, maxNX=0, maxNY=0, linestyle=None, powerLimits=[-4, 2, -4, 2], titlefsize=12, xylabelfsize=12, xytick-fsize=10, labelfsize=10, drawGrid=True, xScientific=False, yScientific=False, yIn-vert=False, xInvert=False, xIsDate=False, xTicks=None, xtickRotation=0, markers=[], markevery=None*)
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

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)

> **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, 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*)
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)
>
> 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)

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*)
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

Returns:

the axis object for the plot

Raises:

No exception is raised.

plotArray (*plotnum*, *inarray*, *slicedim=0*, *subtitles=None*, *xlabel=None*, *maxNX=0*, *maxNY=0*, *titlefsize=8*, *xylabelfsize=8*, *xytickfsize=8*)
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): subplot number, 1-based index

inarray (array): np.array

slicedim (int): slice along columns (0) or rows (1)

subtitles (list): a list of strings as subtitles for each subplot

xlabel (string): x-axis label (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)

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*)

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)
>
> **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 0x05105E90>, linewidth=0.1, azim=45, elev=30*)

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): camera azimuth angle viewing the graph [degrees]
>
> elev (float): camera evelation angle viewing the graph [degrees]

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 0x05105E90>, 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=False, negativeSolid=False, contLabel=False, contFmt='%.2f', contCol='k', contFonSz=8, contLinWid=0.5*)
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 (bool): draw the contour at zero

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)

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, 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*)
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)

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)

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, 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*)
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)

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)

Returns:

the axis object for the plot

Raises:

No exception is raised.

showImage (*plotnum*, *img*, *ptitle=None*, *cmap=<matplotlib.colors.LinearSegmentedColormap object at 0x051051F0>*, *titlefsize=12*, *cbarshow=False*, *cbarorientation='vertical'*, *cbarcustomticks=[ ]*, *cbarfontsize=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)

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)

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, markeredge-color=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, cbarfor-mat='.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.

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:

```python
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.

SPHERICAL PLOTTING UTILITY

5.1 Overview

This module provides tools for creating and viewing spherical plots.

The spherical plotting tool, using Mayavi, requires two sets of data in order to create the spherical plot: the vertex locations in (x,y,z) and the spatial relationship between the vertices, i.e. triangulation of nearest neighbours. This spatial relationship is required to create surface elements between the vertices. If the spatial relationship is not known, the data is merely a cloud of points, with no surface content.

The easiest way to create the spatial relationships between the vertices was to use a complex hull polygon model of an object. The polygon or wireframe model has the requires vertices and spatial relationships.

In the original application of this tool, a series of spheres were created using MeshLab. The essential property of these spheres was that the vertices on the surface of the spheres were spaced equidistant over the surface, i.e. an optimal spatial sampling distribution. The files were exported as OFF files, and should be in the pyradi data/plotspherical directory. There are 6 possible input files, each with different number of samples on the unit sphere: 12, 42, 162, 642, 2562 or 10242.

Any object, with the required vertices and spatial relationships can be used. it does not have to be equi-sampled spheres.

Note that the spherical plot has no way to discriminate between negative values and a pi phase shift: there is confusion between sign and direction. This is inherent in the conversion between cartesian and spherical coordinates. The user has to make provision for this, possibly by plotting only negative or only positive values.

The data must be in the OFF wireframe format.

There are two possible trajectory file types:

- 'Rotate' Stationary sensor and object with the target rotating. In this case the trajectory file specifies the target trajectory.

- 'Orbit' Stationary object with orbiting sensor. In this case the trajectory file specifies the sensor trajectory.

The sphere data available in pyradi/data/plotspherical are:

Filename .	Resolution (degrees)	Number points	Number triangles
sphere_0_12	63.4	12	20
sphere_1_42	33.9	42	80
sphere_2_162	17.2	162	320
sphere_3_642	8.6	642	1280
sphere_4_2562	4.32	2562	5120
sphere_5_10242	2.16	10242	20480
sphere_6_40962	1.08	40962	81920
sphere_7_163842	0.54	163842	327680

The workflow is as follows:

1. Use writeOSSIMTrajOFFFile (or your own equivalent) to calculate the appropriate trajectory file. At the same time, there are two additional files created (vertices and triangles) - keep these safe.

2. Create your data set (e.g. run simulation) using the trajectory file. Collect the simulation data in a format for plotting.

3. Use the simulation data, together with the triangles and vertices file, to plot the data. The triangles and vertices are require to set up the plotting environment, consider this the three-dimensional 'grid', while the simulation data provides the data to be plotted in this grid.

This tool was originally developed to create trajectory files for the Denel/CSIR OSSIM simulation. The code was restructured for greater universal application, but the final example is still an OSSIM case.

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

5.2 Module functions

pyradi.ryplotspherical.**readOffFile**(*filename*)
> Reads an OFF file and returns the vertices and triangles in numpy arrays.

> The OFF file is read and the data captured in the array structures. This is a fairly trivial reading task.

> **Args:**

>> filename (string): name of the OFF file

> **Returns:**

>> vertices(np.array([])): array of vertices as [x y z]
>> triangles(np.array([])): array of triangles as []

> **Raises:**

>> No exception is raised.

pyradi.ryplotspherical.**getRotateFromOffFile**(*filename*, *xPos*, *yPos*, *zPos*)
> Reads an OFF file and returns object attitude and position.

> Calculate the pitch and yaw angles to point the object's X-axis towards the OFF file vertex directions.

> Euler order is yaw-pitch-roll, with roll equal to zero. Yaw is defined in xy plane. Pitch is defined in xz plane. Roll is defined in yz plane.

> The object is assumed to stationary at the position (xPos, yPos, zPos), the position arrays are the same length as the attitude angle arrays, but all values in each individual array are all the same.

> **Args:**

>> filename (string): OFF file filename
>> xPos (double): object position on x axis
>> yPos (double): object position on y axis
>> zPos (double): object position on z axis

> **Returns:**

>> x(np.array()): array of x object location values
>> y(np.array()): array of y object location values
>> z(np.array()): array of z object location values
>> roll(np.array()): array of object location roll values
>> pitch(np.array()): array of object location pitch values

yaw(np.array()): array of object location yaw values

vertices(np.array([])): array of vertices as [x y z]

triangles(np.array([])): array of triangles as []

Raises:

No exception is raised.

pyradi.ryplotspherical.**getOrbitFromOffFile**(*filename*, *xTargPos*, *yTargPos*, *zTargPos*, *distance*)

Reads an OFF file and returns sensor attitude and position.

Calculate the sensor attitude and position such that the sensor always look at the object located at (xTargPos, yTargPos, zTargPos), at a constant distance.

Euler order is yaw-pitch-roll, with roll equal to zero. Yaw is defined in xy plane. Pitch is defined in xz plane. Roll is defined in yz plane.

The object is assumed to stationary at the position (xTargPos, yTargPos, zTargPos).

Args:

filename (string): OFF file filename

xTargPos (double): x target object position (fixed)

yTargPos (double): y target object position (fixed)

zTargPos (double): z target object position (fixed)

distance (double): range at which sensor orbits the target

Returns:

x(np.array()): array of x sensor position values

y(np.array()): array of y sensor position values

z(np.array()): array of z sensor position values

roll(np.array()): array of sensor roll values

pitch(np.array()): array of sensor pitch values

yaw(np.array()): array of sensor yaw values

vertices(np.array([])): array of vertices as [x y z]

triangles(np.array([])): array of triangles as []

Raises:

No exception is raised.

pyradi.ryplotspherical.**writeOSSIMTrajOFFFile**(*filename*, *trajType*, *distance*, *xTargPos*, *yTargPos*, *zTargPos*, *xVel*, *yVel*, *zVel*, *engine*, *deltaTime*)

Reads OFF file and create OSSIM trajectory files for rotating object or orbiting sensor.

This function writes a file in the custom OSSIM trajectory file format. Use this function as an example on how to use the ryplotspherical functionality in your application.

Two different types of trajectory files are created:

1. **trajType = 'Rotate'** Calculate attitude (pitch and yaw angles only, roll is zero) to orientate an object's x-axis along the vertices in the OFF file. The location of the object is fixed at (xTarg-Pos,yTargPos,zTargPos).

2. **trajType = 'Orbit'** Calculate location and attitude (pitch and yaw angles only, roll is zero) of an orbiting sensor looking a a fixed location (xTargPos, TargPos, zTargPos) from a given distance.

The velocity and engine settings are constant for all views at the values specified.

The deltaTime parameter is used to define the time increment to be used in the trajectory file.

Two additional files are also written to assist with the subsequent viewing.

1. The **vertex** file contains the normalised direction vectors between the object and observer. Depending on the trajectory type (see above), the sensor and object switch locations for these vectors. These vectors are the directions of sampled intensity values.

2. The **triangles** file defines triangles that provides the spatial linking between adjacent vectors, used when plotting the data. We plot the complex hull comprising these triangles, with vertices along the direction vectors, with length given by the simulated data set.

OSSIM Users: For examples of how to use these trajectory files, see test points tp01l (that is lowercase L) and tp01m. The scenario files are present in the the appropriate test point directory (l and m) and the plotting routines are in the tp01 utils directory.

Args:

> filename (string): OFF file filename
>
> trajType (string): type of trajectory: 'Rotate' or 'Orbit'
>
> distance (double): distance from sensor to object
>
> xTargPos (double): object x position.
>
> yTargPos (double): object y position.
>
> zTargPos (double): object z position.
>
> xVel (double): velocity in x direction
>
> yVel (double): velocity in y direction
>
> zVel (double): velocity in z direction
>
> engine (double): engine settiing
>
> deltaTime (double): sampling time increment in output file

Returns:

> writes a trajectory file
>
> writes a triangles file
>
> writes a vertices file

Raises:

> No exception is raised.

pyradi.ryplotspherical.**writeOSSIMTrajElevAzim**(*numSamplesAz*, *filename*, *trajType*, *distance*, *xTargPos*, *yTargPos*, *zTarg-Pos*, *xVel*, *yVel*, *zVel*, *engine*, *delta-Time*)

Create OSSIM trajectory files for rotating object or orbiting sensor for constant increments in azimuth and elevation (yaw and pitch).

This function writes a file in the custom OSSIM trajectory file format. Use this function as an example on how to use the ryplotspherical functionality in your application.

Two different types of trajectory files are created:

1. **trajType = 'Rotate'** Calculate attitude (pitch and yaw angles only, roll is zero) to orientate an object's x-axis along the elevation and azimuth vectors. The location of the object is fixed at (xTargPos,yTargPos,zTargPos).

2. **trajType = 'Orbit'** Calculate location and attitude (pitch and yaw angles only, roll is zero) of an orbiting sensor looking a a fixed location (xTargPos, TargPos, zTargPos) from a given distance along the elevation and azimuth vectors.

The velocity and engine settings are constant for all views at the values specified.

The deltaTime parameter is used to define the time increment to be used in the trajectory file.

Two additional files are also written to assist with the subsequent viewing.

1. The **azimuth** file contains the sample values around the equator. The value ranges from zero to two*pi

2. The **elevation** file contains the sample values from North pole to South pole. The value ranges from -pi to +pi

Args:

> numSamplesAz (int): The number of samples along the equator
> filename (string): output file filename
> trajType (string): type of trajectory: 'Rotate' or 'Orbit'
> distance (double): distance from sensor to object
> xTargPos (double): object x position.
> yTargPos (double): object y position.
> zTargPos (double): object z position.
> xVel (double): velocity in x direction
> yVel (double): velocity in y direction
> zVel (double): velocity in z direction
> engine (double): engine settiing
> deltaTime (double): sampling time increment in output file

Returns:

> writes a trajectory file
> writes a azimuth, elevation file

Raises:

> No exception is raised.

pyradi.ryplotspherical.**getOrbitFromElevAzim**(*azimuth*, *elevation*, *xTargPos*, *yTargPos*, *zTargPos*, *distance*)

Reads an OFF file and returns sensor attitude and position.

Calculate the sensor attitude and position such that the sensor always look at the object located at (xTargPos, yTargPos, zTargPos), at a constant distance.

Euler order is yaw-pitch-roll, with roll equal to zero. Yaw is defined in xy plane. Pitch is defined in xz plane. Roll is defined in yz plane.

The object is assumed to stationary at the position (xTargPos, yTargPos, zTargPos).

Args:

> azimuth (np.array(N,)): azimuth values
> elevation (np.array(N,)): azimuth values
> filename (string): OFF file filename
> xTargPos (double): x target object position (fixed)
> yTargPos (double): y target object position (fixed)
> zTargPos (double): z target object position (fixed)
> distance (double): range at which sensor orbits the target

Returns:

> x(np.array()): array of x sensor position values
> y(np.array()): array of y sensor position values
> z(np.array()): array of z sensor position values
> roll(np.array()): array of sensor roll values
> pitch(np.array()): array of sensor pitch values
> yaw(np.array()): array of sensor yaw values
> azel(np.array()): array of azimuth,elevation values for each sample

Raises:

> No exception is raised.

pyradi.ryplotspherical.**getRotateFromElevAzim**(*azimuth*, *elevation*, *xPos*, *yPos*, *zPos*)

Reads an OFF file and returns object attitude and position.

5.2. Module functions 41

Calculate the pitch and yaw angles to point the object's X-axis towards the OFF file vertex directions.

Euler order is yaw-pitch-roll, with roll equal to zero. Yaw is defined in xy plane. Pitch is defined in xz plane. Roll is defined in yz plane.

The object is assumed to stationary at the position (xPos, yPos, zPos), the position arrays are the same length as the attitude angle arrays, but all values in each individual array are all the same.

Args:

> azimuth (np.array(N,)): azimuth values
>
> elevation (np.array(N,)): azimuth values
>
> xPos (double): object position on x axis
>
> yPos (double): object position on y axis
>
> zPos (double): object position on z axis

Returns:

> x(np.array()): array of x object location values
>
> y(np.array()): array of y object location values
>
> z(np.array()): array of z object location values
>
> roll(np.array()): array of object location roll values
>
> pitch(np.array()): array of object location pitch values
>
> yaw(np.array()): array of object location yaw values
>
> azel(np.array()): array of azimuth,elevation values for each sample

Raises:

> No exception is raised.

pyradi.ryplotspherical.**plotSpherical**(*figure*, *dataset*, *vertices*, *triangles*, *ptitle=u''*, *tsize=0.4*, *theight=1*)
Plot the spherical data given a data set, triangle set and vertex set.

The vertex set defines the direction cosines of the individual samples. The triangle set defines how the surrface must be structured between the samples. The data set defines, for each direction cosine, the length of the vector.

Args:

> figure(int): mlab figure number
>
> dataset(np.array(double)): array of data set values
>
> vertices(np.array([])): array of direction cosine vertices as [x y z]
>
> triangles(np.array([])): array of triangles as []
>
> ptitle(string): title or header for this display
>
> tsize(double): title width in in normalised figure width
>
> theight(double): title top vertical location in normalised figure height

Returns:

> provides an mlab figure.

Raises:

> No exception is raised.

pyradi.ryplotspherical.**plotOSSIMSpherical**(*basefigure*, *nColours*, *plottitle*, *datafile*, *vertexfile*, *trianglefile*)
Plot the spherical data given a data set, triangle set and vertex set.

The vertex set defines the direction cosines of the individual samples. The triangle set defines how the surrface must be structured between the samples. The data set defines, for each direction cosine, the length of the vector.

There is no means to discriminate between negative and pi phase shift. In this function we plot colour ratio values initially in absolute form, then only positive and then only negative values. In between these two shells the values are going through zero.

Args:

> basefigure (int): value where figure count must start
>
> nColours ([int]): selection of colours to display
>
> plottitle (string): plot title or header
>
> datafile (string): dataset file filename
>
> vertexfile (string): vertex file filename
>
> trianglefile (string): triangles file filename

Returns:

> provides an mlab figure.

Raises:

> No exception is raised.

`pyradi.ryplotspherical.`**`sphericalPlotElevAzim`**(*figure*, *azimuth*, *elevation*, *value*, *ptitle=None*, *colormap=u'Spectral'*, *doWireFrame=False*, *line_width=0.2*)
Plot spherical data (azimuth, elevation and value) given in spherical format

This function assumes that the polar data is defined in terms of elevation angle with zero at the equator and azimuth measured around the equator, from 0 to 2pi. All angles are in degrees.

All axes, x, y, and z are scaled with the magnitude of value in the (azim, elev) direction.

The colour on the surface represents the value at the given azim/elev location.

Args:

> figure (int): mlab figure number
>
> azimuth (numpy 1D array): vector of values
>
> elevation (numpy 1D array): vector of values
>
> value (numpy 2D array): array with values corresponding with azim/elevation
>
> ptitle (string): plot title
>
> colormap (string): defines the colour map to be used
>
> doWireFrame (bool): switches fireframe on or off
>
> line_width (double): wireframe line width

Returns:

> provides an mlab figure.

Raises:

> No exception is raised.

`pyradi.ryplotspherical.`**`polarPlotElevAzim`**(*figure*, *azimuth*, *elevation*, *value*, *ptitle=None*, *colormap=u'Spectral'*, *doWireFrame=False*, *line_width=0.2*)
Plot spherical data (azimuth, elevation and value) given in polar format.

This function assumes that the polar data is defined in terms of elevation angle with zero at the equator and azimuth measured around the equator, from 0 to 2pi. All angles are in degrees.

The x and y axes are scaled with the maximum magnitude of value. The z axis is scaled with the actual magnitude of value in the (azim, elev) direction.

The colour on the surface represents the value at the given azim/elev location.

Args:

> figure (int): mlab figure number

azimuth (numpy 1D array): vector of values

elevation (numpy 1D array): vector of values

value (numpy 2D array): array with values corresponding with azim/elevation

ptitle (string): plot title

colormap (string): defines the colour map to be used

doWireFrame (bool): switches fireframe on or off

line_width (double): wireframe line width

Returns:

provides an mlab figure.

Raises:

No exception is raised.

`pyradi.ryplotspherical.`**`globePlotElevAzim`**(*figure*, *azimuth*, *elevation*, *value*, *ptitle=None*,
colormap=u'Spectral', *doWireFrame=False*,
line_width=0.2)

Plot spherical data on a sphere.

This function assumes that the polar data is defined in terms of elevation angle with zero at the equator and azimuth measured around the equator, from 0 to 2pi. All angles are in degrees.

The x, y, and z values defines vertices on a sphere. The colour on the sphere represents the value at the given azim/elev location.

Args:

figure (int): mlab figure number

azimuth (numpy 1D array): vector of values

elevation (numpy 1D array): vector of values

value (numpy 2D array): array with values corresponding with azim/elevation

ptitle (string): plot title

colormap (string): defines the colour map to be used

doWireFrame (bool): switches fireframe on or off

line_width (double): wireframe line width

Returns:

provides an mlab figure.

Raises:

No exception is raised.

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 functions

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

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].

Returns:

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

Raises:

No exception is raised.

`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.

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 0x04DE4630>*)

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.**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 rangefinder 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 lookup

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

rangeGuess (float): starting value range estimate

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

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.**detectThresholdToNoise**(*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.**detectSignalToNoise**(*ThresholdToNoise*, *pD*)

Solve for 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.

PTW FILE FUNCTIONS

7.1 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

7.2 Module classes

class pyradi.ryptw.**JadeCalibrationData**(*filename*, *datafileroot*)
> Container to describe the calibration data of a Jade camera.

> **CalculateCalibrationTables**()
> > Calculate the mapping functions between digital level, 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.

> **LoadPrepareData**()
> > Load the camera calibration data from files and preprocess spectrals data

> **LookupDLRad**(*DL*, *Tint*)
> > Calculate the radiance associated with a DL and Tint pair. Interpolate linearly on Tint radiance not temperature.

> **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.
>
> **LookupDLTemp**(*DL*, *Tint*)
>> Calculate the temperature associated with a DL and Tint pair. Here we interpolate linearly on Tint temperature - actually we must interpolate linearly on radiance - to be done later. Note that dicLookupRadTemp is available for both Tint values, but it has the same value in both cases.
>
> **PlotDLRadiance**()
>> Plot DL level versus radiance for both camera temperatures
>
> **PlotDLTemp**()
>> Plot digital level versus temperature for both camera temperatures
>
> **PlotRadiance**()
>> Plot all spectral radiance data for the calibration temperatures.
>
> **PlotSpectrals**()
>> Plot all spectral curve data to a single graph.
>
> **PlotTempRadiance**()
>> Plot temperature versus radiance for both camera temperatures
>
> **PlotTintRad**()
>> Plot optics radiance versus instrument temperature
>
> **Print**()
>> Write the calibration data file data to a string and return string.

class `pyradi.ryptw.`**PTWFrameInfo**
> Class to store the ptw file header information.

7.3 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.

MODTRAN UTILITY

8.1 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

8.2 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. 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']

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.

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.

THREE-DIMENSIONAL NOISE CALCULATION

9.1 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

9.2 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 sparial 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.

COLOUR COORDINATES

10.1 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

10.2 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.

BULK DETECTOR MODELLING

11.1 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

11.2 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:

11.2. Module functions 65

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]

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/.

12.1 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

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.

FOURTEEN

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.

p