pyradi Documentation

Release

pyradi team

March 05, 2013

CONTENTS

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

The toolkit is available at http://code.google.com/p/pyradi

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

CHAPTER

ONE

INTRODUCTION

1.1 Overview

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

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

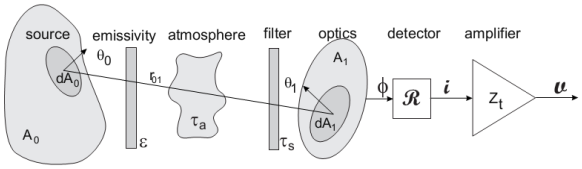
1.2 Toolkit approach

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

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

1.3 Example application

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

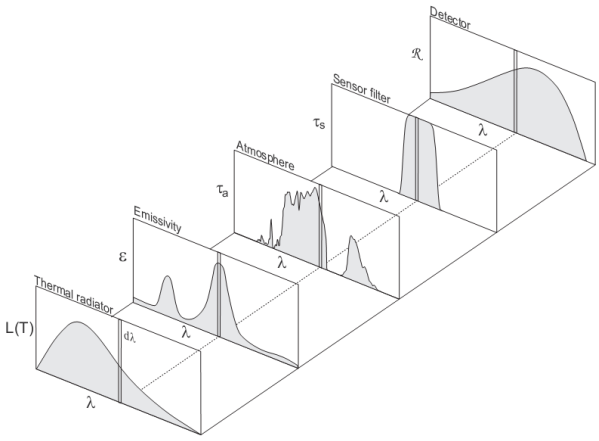


The amplifier output signal can be calculated in the following equation, by integrating over all wavelengths, over the full source area A_0 and over the optical aperture area A_1,

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

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

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



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

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

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

This example is a relatively complete worked example. The objective is to calculate the signal of a simple sensor, detecting the presence or absence of a flame in the sensor field of view. The sensor is pointed to an area just outside

a furnace smokestack, against a clear sky background. The sensor must detect a change in signal, to indicate the presence or absence of a flame.

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

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

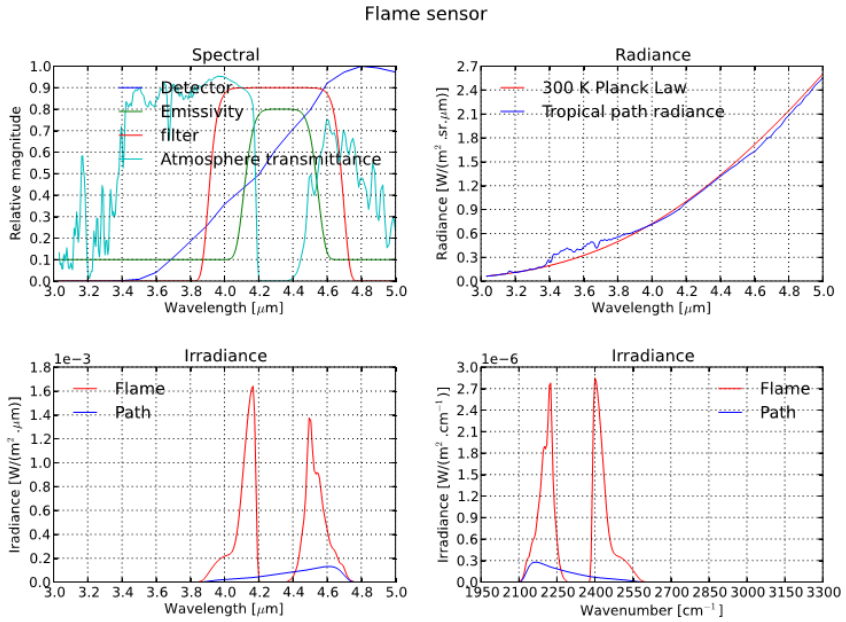
The distance between the flame and the sensor is 1000~m. The atmospheric properties are calculated with the Modtran Tropical climatic model. The path is oriented such that the sensor stares out to space, at a zenith angle of 88°. The spectral transmittance and path radiance along this path is shown in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

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

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

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

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



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

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

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

CHAPTER

TWO

PLANCK AND THERMAL RADIATION

2.1 Overview

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

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.

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. Exitance can be given in radiant or photon rate units, depending on user input in type.

Args:

spectral (np.array[N,]): spectral vector.
temperature (float): Temperature scalar 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:

(np.array[N,]): 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 vector of -1 values if illegal type is requested.

Raises:

No exception is raised.

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

Args:

spectral (np.array[N,]): spectral vector in [micrometer], [cm-1] or [Hz].
temperature (float): Temperature scalar 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:

(np.array[N,]): 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 vector of -1 values if illegal type is requested.

Raises:

No exception is raised.

pyradi.ryplanck.**stefanboltzman**(*temperature*, *type=u'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:

temperature (float): temperature scalar 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**(*wavelength*, *temperature*)
Planck function in wavelength for radiant exitance.

Args:

wavelength (np.array[N,]): wavelength vector in [um]
temperature (float): Temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

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

Args:

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

Returns:

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

Raises:

No exception is raised.

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

Args:

frequency (np.array[N,]): frequency vector in [Hz]
temperature (float): Temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

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

Args:

wavelength (np.array[N,]): wavelength vector in [um]
temperature (float): temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

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

Args:

wavenumber (np.array[N,]): wavenumber vector in [cm^-1]
temperature (float): temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

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

Args:

frequency (np.array[N,]): frequency vector in [Hz]
temperature (float): Temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

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

Args:

wavelength (np.array[N,]): wavelength vector in [um]
temperature (float): temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

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

Args:

wavenumber (np.array[N,]): wavenumber vector in [cm^-1]
temperature (float): temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

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

Args:

frequency (np.array[N,]): frequency vector in [Hz]
temperature (float): temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

pyradi.ryplanck.**dplnckql**(*wavelength*, *temperature*)

Temperature derivative of Planck function in wavenumber domain for radiance exitance.

Args:

wavelength (np.array[N,]): wavelength vector in [um]
temperature (float): temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

pyradi.ryplanck.**dplnckqn**(*wavenumber*, *temperature*)

Temperature derivative of Planck function in wavenumber domain for photon rate.

Args:

wavenumber (np.array[N,]): wavenumber vector in [cm^-1]
temperature (float): temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

pyradi.ryplanck.**dplnckqf**(*frequency*, *temperature*)

Temperature derivative of Planck function in frequency domain for photon rate.

Args:

frequency (np.array[N,]): frequency vector in [Hz]
temperature (float): temperature scalar in [K]

Returns:

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

Raises:

No exception is raised.

CHAPTER

THREE

FILE READING/WRITING UTILITY

3.1 Overview

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

See the __main__ function for examples of use.

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*)

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

This function loads column data from a text file, manipulating the data read in. The individual vector data must be given in columns in the file, with the abscissa (x-value) in first column (col 0 in Python) and any number of ordinate (y-value) vectors in second and later columns.

Note: leave only single separators (e.g. spaces) between columns! Also watch out for a single sapce 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.

Returns:

(np.array[N,M]): The interpolated, M columns of N rows, processed array.

Raises:

No exception is raised.

pyradi.ryfiles.**loadHeaderTextFile**(*filename, loadCol=[1], comment=None*)
Loads column data from a text file, using the csv package.

Using the csv package, loads column header data from a file, from the firstrow. 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*)
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?

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 (numpy.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 (numpy.ndarray): vartype numpy array of dimensions (frames,rows,cols),
None if error occurred

Raises:

No exception is raised.

CHAPTER

FOUR

PLOTTING UTILITY

4.1 Overview

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

See the __main__ function for examples of use.

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 radiometry plots.

This class provides a wrapper around Matplotlib to provide a plotting environment specialised towards radiometry results. These functions were developed to provide well labelled plots by entering only one or two lines.

Provision is made for plots containing subplots (i.e. multiple plots on the same figure), linear scale and log scale plots, and cartesian and polar plots. Simple 3D line plots can also be made.

buildPlotCol(*plotCol*, *n*)

Returns 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.

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 plot

Args:

None

Returns:

A handle to the current plot.

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.

Raises:

No exception is raised.

logLog(*plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], label=[], legendAlpha=0.0, pltaxis=None, maxNX=10, maxNY=10, powerLimits=[-4, 2, -4, 2], titlefsize=12*)

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
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 line style, list with M entries, use default if [] (optional)
label ([strings]): legend label for ordinate, list with M entries (optional)
legendAlpha (float): transparancy for legend (optional)
pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. default if all zeros. (optional)
maxNX (int): draw maxNX+1 tick labels on x axis (optional)
maxNY (int): draw maxNY+1 tick labels on y axis (optional)
powerLimits[float]: axis notation power limits [x-neg, x-pos, y-neg, y-pos] (optional)
titlefsize (int): title font size, default 12pt (optional)

Returns:

Nothing

Raises:

No exception is raised.

myPlot(*plotcommand, plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], label=[], legendAlpha=0.0, pltaxis=None, maxNX=0, maxNY=0, powerLimits=[-4, 2, -4, 2], titlefsize=12*)

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.

Args:

plotcommand: name of a MatplotLib plotting function
plotnum (int): subplot number
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 line style, list with M entries, use default if []
label ([strings]): legend label for ordinate, list with M entries
legendAlpha (float): transparancy for legend
pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. default if all zeros.
maxNX (int): draw maxNX+1 tick labels on x axis
maxNY (int): draw maxNY+1 tick labels on y axis
powerLimits[float]: axis notation power limits [x-neg, x-pos, y-neg, y-pos]
titlefsize (int): title font size, default 12pt (optional)

Returns:

Nothing

Raises:

No exception is raised.

plot(*plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], label=[], legendAlpha=0.0, pltaxis=None, maxNX=10, maxNY=10, powerLimits=[-4, 2, -4, 2], titlefsize=12*)
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
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 line style, list with M entries, use default if [] (optional)
label ([strings]): legend label for ordinate, list with M entries (optional)
legendAlpha (float): transparancy for legend (optional)
pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. default if all zeros. (optional)
maxNX (int): draw maxNX+1 tick labels on x axis (optional)
maxNY (int): draw maxNY+1 tick labels on y axis (optional)
powerLimits[float]: axis notation power limits [x-neg, x-pos, y-neg, y-pos]
titlefsize (int): title font size, default 12pt (optional)

Returns:

Nothing

Raises:

No exception is raised.

plot3d(*plotnum*, *x*, *y*, *z*, *ptitle=None*, *xlabel=None*, *ylabel=None*, *zlabel=None*, *plotCol=[ ]*, *label=None*, *legendAlpha=0.0*, *titlefsize=12*)

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 vertices, each column representing a different function 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
x (np.array[N,] or [N,M]): x coordinates of vertices
y (np.array[N,] or [N,M]): y coordinates of vertices
z (np.array[N,] or [N,M]): z coordinates of vertices
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 line style, list with M entries, use default if [] (optional)
label ([strings]): legend label for ordinate, list with M entries (optional)
legendAlpha (float): transparancy for legend (optional)
titlefsize (int): title font size, default 12pt (optional)

Returns:

Nothing

Raises:

No exception is raised.

plotMarkers(*plotnum*)

Add markers to the subplot

Args:

plotnum (int): subplot number
theta (np.array[N,] or [N,1]): vector of angular values
radial (np.array[M,] or [M,1]): vector if radial values
zvals (np.array[N,M]): values on a (theta,radial) 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

Returns:

Nothing

Raises:

No exception is raised.

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

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.

Args:

plotnum (int): subplot number
theta (np.array[N,] or [N,1]): angular abscissa
r (np.array[N,] or [N,M]): radial ordinates - could be M columns
ptitle (string): plot title (optional)
plotCol ([strings]): plot line style, list with M entries, use default if [] (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 be highlighted (optional)
highlightCol (string): highlighted colour string (optional)
highlightWidth (int): highlighted line width(optional)
legendAlpha (float): transparancy for legend (optional)
rscale ([rmin, rmax]): plotting limits. default if all 0 (optional)
rgrid ([rinc, rmax]): radial grid default if all 0. if rinc=0 then rmax is number of ntervals. (optional)
thetagrids (float): theta grid interval [degrees] (optional)
direction (string)= 'counterclockwise' or 'clockwise' (optional)
zerooffset (float) = rotation offset where zero should be [rad] (optional)
titlefsize (int): title font size, default 12pt (optional)

Returns:

Nothing

Raises:

No exception is raised.

polar3d(*plotnum, theta, radial, zvals, ptitle=None, xlabel=None, ylabel=None, zlabel=None, zscale=None, titlefsize=12, thetaStride=1, radialstride=1, meshCmap=<matplotlib.colors.LinearSegmentedColormap object at 0x056A72F0>*)

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

theta (np.array[N,] or [N,1]): vector of angular values
radial (np.array[M,] or [M,1]): vector if radial values
zvals (np.array[N,M]): values on a (theta,radial) 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)
thetaStride (int): theta stride in input data (optional)
radialstride (int): radial stride in input data (optional)
meshCmap (cm): color map for the mesh (optional)

Returns:

Nothing

Raises:

No exception is raised.

polarMesh(*plotnum*, *theta*, *radial*, *zvals*, *ptitle=None*, *shading=u'flat'*, *radscale=None*, *titlefsize=12*, *meshCmap=<matplotlib.colors.LinearSegmentedColormap object at 0x056A72F0>*, *cbarshow=False*, *cbarorientation=u'vertical'*, *cbarcustomticks=[]*, *cbarfontsize=12*)

Polar colour 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
theta (np.array[N,] or [N,1]): vector of angular values
radial (np.array[M,] or [M,1]): vector if radial values
zvals (np.array[N,M]): values on a (theta,radial) 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

Returns:

Nothing

Raises:

No exception is raised.

saveFig(*filename=u'mpl.png'*, *dpi=100*, *bbox_inches=u'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.

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=[], label=[], legendAlpha=0.0, pltaxis=None, maxNX=10, maxNY=10, powerLimits=[-4, 2, -4, 2], titlefsize=12*)

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
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 line style, list with M entries, use default if [] (optional)
label ([strings]): legend label for ordinate, list with M entries (optional)
legendAlpha (float): transparancy for legend (optional)
pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. default if all zeros. (optional)
maxNX (int): draw maxNX+1 tick labels on x axis (optional)
maxNY (int): draw maxNY+1 tick labels on y axis (optional)
powerLimits[float]: axis notation power limits [x-neg, x-pos, y-neg, y-pos] (optional)
titlefsize (int): title font size, default 12pt (optional)

Returns:

Nothing

Raises:

No exception is raised.

semilogY(*plotnum, x, y, ptitle=None, xlabel=None, ylabel=None, plotCol=[], label=[], legendAlpha=0.0, pltaxis=None, maxNX=10, maxNY=10, powerLimits=[-4, 2, -4, 2], titlefsize=12*)

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
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 line style, list with M entries, use default if [] (optional)
label ([strings]): legend label for ordinate, list withM entries (optional)
legendAlpha (float): transparancy for legend (optional)
pltaxis ([xmin, xmax, ymin,ymax]): scale for x,y axes. default if all zeros. (optional)
maxNX (int): draw maxNX+1 tick labels on x axis (optional)
maxNY (int): draw maxNY+1 tick labels on y axis (optional)
powerLimits[float]: axis notation power limits [x-neg, x-pos, y-neg, y-pos] (optional)
titlefsize (int): title font size, default 12pt (optional)

Returns:

Nothing

Raises:

No exception is raised.

showImage(*plotnum, img, ptitle=None, cmap=<matplotlib.colors.LinearSegmentedColormap object at 0x0569E650>, titlefsize=12, cbarshow=False, cbarorientation=u'vertical', cbarcustomticks=[], cbarfontsize=12*)

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

Args:

plotnum (int): subplot number
img (np.ndarray): numpy 2d array
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 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

Returns:

Nothing

Raises:

No exception is raised.

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

Filled marker user-settable values.

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

Values relevant to filled makers are as follows:

marker = ['o', 'v', '^', '<', '>', '8', 's', 'p', '*', 'h', 'H', 'D', 'd']

fillstyle = ['full', 'left', 'right', 'bottom', 'top', 'none']

colour names = http://www.w3schools.com/html/html_colornames.asp

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

Collect maker 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.

CHAPTER

FIVE

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
sphere_0_12	56.9	12
sphere_1_42	28.5	42
sphere_2_162	14.2	162
sphere_3_642	7.1	642
sphere_4_2562	3.56	2562
sphere_5_10242	1.78	10242

The workflow is as follows:

1. Use writeRotatingTargetOssimTrajFile (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.

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(numpy.array([])): array of vertices as [x y z]
triangles(numpy.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): scale factor to be applied to x axis
yPos (double): scale factor to be applied to y axis
zPos (double): scale factor to be applied to z axis

Returns:

x(numpy.array()): array of x values
y(numpy.array()): array of y values
z(numpy.array()): array of z values
roll(numpy.array()): array of roll values
pitch(numpy.array()): array of pitch values
yaw(numpy.array()): array of yaw values
vertices(numpy.array([])): array of vertices as [x y z]
triangles(numpy.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(numpy.array()): array of x values
y(numpy.array()): array of y values
z(numpy.array()): array of z values
roll(numpy.array()): array of roll values
pitch(numpy.array()): array of pitch values
yaw(numpy.array()): array of yaw values
vertices(numpy.array([])): array of vertices as [x y z]
triangles(numpy.array([])): array of triangles as []

Raises:

No exception is raised.

pyradi.ryplotspherical.**plotSpherical**(*dataset*, *vertices*, *triangles*, *ptitle=u''*, *tsize=0.4*, *theight=0.95*)

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 surfrace must be structured between the samples. The data set defines, for each direction cosine, the length of the vector.

Args:

dataset(numpy.array(double)): array of data set values
vertices(numpy.array([])): array of direction cosine vertices as [x y z]
triangles(numpy.array([])): array of triangles as []
ptitle(string): title or header for this display
tsize(double): title size (units not quite clear)
theight(double): title height (y value) (units not quite clear)

Returns:

provides and mlab figure.

Raises:

No exception is raised.

pyradi.ryplotspherical.**writeRotatingTargetOssimTrajFile**(*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 (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.

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 **directions** 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 ro 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.**plotOSSIMSpherical**(*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 surfrace 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:

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 and mlab figure.

Raises:

No exception is raised.

CHAPTER

SIX

UTILITY FUNCTIONS

6.1 Overview

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

See the __main__ function for examples of use.

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=u'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=u''*)

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=u''*)
Convert spectral density quantities, i.e. between W/(m^2.um), W/(m^2.cm^-1) and W/(m^2.Hz). Return always positive.

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 0x04312570>*)

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 abritrary 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.

CHAPTER

SEVEN

PTW FILE FUNCTIONS

7.1 Overview

This module provides functions to read the contents of files in the PTW file format.

Callable functions :

readPTWHeader(ptwfilename)
showHeader(header)
getPTWFrame (s, frameindex)

readPTWHeader(ptwfilename) : Returns a class object defining all the header information

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 author wishes 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.

7.2 Module functions

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

Read a PTW file header

Args:

filename (string) with full path

Returns:

Header (class) containing all PTW header information

Raises:

No exception is raised.

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

Utility function to display the PTW header information

Args:

header (class object)

Returns:

None

Raises:

No exception is raised.

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

Retrieve a single PTW frame

Args:

header (class object)

frameindex (integer): The frame to be extracted

Returns:

s.data (numpy.ndarray): requested frame DL values, dimensions (rows,cols)

Raises:

No exception is raised.

CHAPTER

EIGHT

MODTRAN UTILITY

8.1 Overview

This module provides rudimentary MODTRAN file reading.

See the __main__ function for examples of use.

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=[ ]*, *delimiter=None*)

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:

numpy.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. 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 between different IEMSCT values. For the most part the differences are hidden in the details, the user does not have to take concern. The various column headers available are as follows:

IEMSCT = 0 has two column name 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 has 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.

pyradi.rymodtran.**savetape7data**(*filename*, *tape7*)

Save the numpy array to a text file. Numeric precision is '%.6e'.

Args:

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

tape7 (numpy.array): an array with the selected columns. Col[0] is the wavenumber.

Returns:

No return.

Raises:

No exception is raised.

CHAPTER

NINE

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.

9.2 Module functions

pyradi.ry3dnoise.**oprDT**(*imgSeq*)
Operator DT averages over frames for each pixel.

Args:

imgSeq (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.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 (numpy.ndarray): numpy array of dimensions (frames,rows,cols)

Returns:

noise (double): total system noise

Raises:

No exception is raised.

CHAPTER

TEN

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.

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.

CHAPTER

ELEVEN

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.

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*, *lx*, *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]

lx: detector depletion layer thickness [m]

theta1: angle between the surface's normal and the radiation in radians

nFront: index of refraction of the material in front of detector

nMaterial: index of refraction of the detector material

Returns:

quantumEffic: spectral quantum efficiency

pyradi.rydetector.**Responsivity**(*wavelength*, *quantumEffic*)

Responsivity quantifies the amount of output seen per watt of radiant optical power input [1]. But, for this application it is interesting to define spectral responsivity that is the output per watt of monochromatic radiation.

The model used here is based on Equations 7.114 in Dereniak's book.

Args:

wavelength: spectral variable [m]
quantumEffic: spectral quantum efficiency

Returns:

responsivity in [A/W]

pyradi.rydetector.**DStar**(*areaDet*, *deltaFreq*, *iNoise*, *responsivity*)

The spectral D* is the signal-to-noise output when 1 W of monochromatic radiant flux is incident on 1 cm2 detector area, within a noise-equivalent bandwidth of 1 Hz.

Args:

areaDet: detector's area in [m2]
deltaFreq: measurement or desirable bandwidth - [Hz]
iNoise: noise current [A]
responsivity: spectral responsivity in [A/W]

Returns

detectivity [cm sqrt[Hz] / W] (note units)

pyradi.rydetector.**NEP**(*iNoise*, *responsivity*)

NEP is the radiant power incident on detector that yields SNR=1 [1].

Args:

iNoise: noise current [A]
responsivity: spectral responsivity in [A/W]

Returns

spectral noise equivalent power [W]

pyradi.rydetector.**Isaturation**(*mobE*, *tauE*, *mobH*, *tauH*, *me*, *mh*, *na*, *nd*, *Eg*, *tDetec*, *areaDet*)

This function calculates the reverse saturation current, by Equation 7.22 in Dereniak's book

Args:

mobE: electron mobility [m2/V.s]
tauE: electron lifetime [s]
mobH: hole mobility [m2/V.s]
tauH: hole lifetime [s]
me: electron effective mass [kg]
mh: hole effective mass [kg]
na: acceptor concentration [m-3]
nd: donor concentration [m-3]
Eg: energy bandgap in [Ev]
tDetec: detector's temperature in [K]
areaDet: detector's area [m2]

Returns:

I0: reverse sat current [A]

pyradi.rydetector.**EgVarshni**(*E0*, *VarshniA*, *VarshniB*, *tempDet*)

This function calculates the bandgap at detector temperature, using the Varshni equation

Args:

E0: band gap at room temperature [eV]
VarshniA: Varshni parameter

VarshniB: Varshni parameter
tempDet: detector operating temperature [K]

Returns:

Eg: bandgap at stated temperature [eV]

pyradi.rydetector.**IXV**(*V*, *IVbeta*, *tDetec*, *iPhoto*, *I0*)

This function provides the diode curve for a given photocurrent.

The same function is also used to calculate the dark current, using IVbeta=1 and iPhoto=0

Args:

V: bias [V]
IVbeta: diode equation non linearity factor;
tDetec: detector's temperature [K]
iPhoto: photo-induced current, added to diode curve [A]
I0: reverse sat current [A]

Returns:

current from detector [A]

pyradi.rydetector.**Noise**(*tempDet*, *IVbeta*, *Isat*, *iPhoto*, *vBias=0*)

This function calculates the noise power spectral density produced in the diode: shot noise and thermal noise

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]

CHAPTER

TWELVE

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

CHAPTER

THIRTEEN

EXAMPLES OF CODE USE

The respective python files all have examples of use and simple test code at the end of each of the files. Execute these python files as scripts to execute the example code, and observe the results.

CHAPTER

FOURTEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[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.

PYTHON MODULE INDEX

p