

---

# **pyrad library reference for developers**

***Release 0.0.1***

**meteoswiss-mdr**

**Jan 10, 2020**



## CONTENTS



Contents:

---



## PYRAD.FLOW.FLOW\_AUX

Auxiliary functions to control the Pyrad data processing flow

**toctree** generated/

`_initialize_listener` `_user_input_listener` `_get_times_and_traj` `_initialize_datasets` `_process_datasets` `_post-process_datasets` `_wait_for_files` `_get_radars_data` `_generate_dataset` `_generate_prod` `_create_cfg_dict` `_create_datacfg_dict` `_create_dscfg_dict` `_create_prdcfg_dict` `_get_datatype_list` `_get_datasets_list` `_get_masterfile_list` `_add_dataset` `_warning_format`

`pyrad.flow.flow_aux._add_dataset (*args, **kwargs)`  
wrapper

### Parameters

**args, kwargs** [arguments] The arguments of the function

### Returns

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

`pyrad.flow.flow_aux._create_cfg_dict (*args, **kwargs)`  
wrapper

### Parameters

**args, kwargs** [arguments] The arguments of the function

### Returns

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

`pyrad.flow.flow_aux._create_datacfg_dict (*args, **kwargs)`  
wrapper

### Parameters

**args, kwargs** [arguments] The arguments of the function

### Returns

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

`pyrad.flow.flow_aux._create_dscfg_dict (*args, **kwargs)`  
wrapper

### Parameters

**args, kwargs** [arguments] The arguments of the function

**Returns**

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

`pyrad.flow.flow_aux._create_prdcfg_dict(*args, **kwargs)`  
wrapper

**Parameters**

**args, kwargs** [arguments] The arguments of the function

**Returns**

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

`pyrad.flow.flow_aux._generate_dataset(*args, **kwargs)`  
wrapper

**Parameters**

**args, kwargs** [arguments] The arguments of the function

**Returns**

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

`pyrad.flow.flow_aux._generate_prod(*args, **kwargs)`  
wrapper

**Parameters**

**args, kwargs** [arguments] The arguments of the function

**Returns**

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

`pyrad.flow.flow_aux._get_datasets_list(*args, **kwargs)`  
wrapper

**Parameters**

**args, kwargs** [arguments] The arguments of the function

**Returns**

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

`pyrad.flow.flow_aux._get_datatype_list(*args, **kwargs)`  
wrapper

**Parameters**

**args, kwargs** [arguments] The arguments of the function

**Returns**

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator



```
pyrad.flow.flow_aux._get_masterfile_list(*args, **kwargs)
    wrapper
```

**Parameters**

**args, kwargs** [arguments] The arguments of the function

**Returns**

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

```
pyrad.flow.flow_aux._get_radars_data(*args, **kwargs)
    wrapper
```

**Parameters**

**args, kwargs** [arguments] The arguments of the function

**Returns**

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

```
pyrad.flow.flow_aux._get_times_and_traj(*args, **kwargs)
    wrapper
```

**Parameters**

**args, kwargs** [arguments] The arguments of the function

**Returns**

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

```
pyrad.flow.flow_aux._initialize_datasets(dataset_levels, cfg, traj=None, infostr=None)
    Initializes datasets. Creates the data set configuration dictionary
```

**Parameters**

**dataset\_levels** [dict] dictionary containing the list of data sets to be generated at each processing level

**cfg** [dict] processing configuration dictionary

**traj** [trajectory object] object containing the trajectory

**infostr** [str] Information string about the actual data processing (e.g. 'RUN57'). This string is added to product files.

**Returns**

**dscfg** [dict] dictionary containing the configuration data for each dataset

**traj** [trajectory object] the modified trajectory object

```
pyrad.flow.flow_aux._initialize_listener()
    initialize the input listener
```

**Returns**

**input\_queue** [queue object] the queue object where to put the quit signal

```
pyrad.flow.flow_aux._postprocess_datasets(dataset_levels, cfg, dscfg, traj=None, infostr=None)
    infostr=None)
```

Processes the radar volumes for a particular time stamp.

### Parameters

**dataset\_levels** [dict] dictionary containing the list of data sets to be generated at each processing level

**cfg** [dict] processing configuration dictionary

**dscfg** [dict] dictionary containing the configuration data for each dataset

**traj** [trajectory object] and object containing the trajectory

**infostr** [str] Information string about the actual data processing (e.g. 'RUN57'). This string is added to product files.

### Returns

**dscfg** [dict] the modified configuration dictionary

**traj** [trajectory object] the modified trajectory object

`pyrad.flow.flow_aux._process_datasets(*args, **kwargs)`  
wrapper

### Parameters

**args, kwargs** [arguments] The arguments of the function

### Returns

**func** [function] The original function if no profiling has to be performed or the function decorated with the memory decorator

`pyrad.flow.flow_aux._user_input_listener(input_queue)`  
Permanently listens to the keyword input until the user types "Return"

### Parameters

**input\_queue** [queue object] the queue object where to put the quit signal

`pyrad.flow.flow_aux._wait_for_files(nowtime, datacfg, datatype_list, last_processed=None)`  
Waits for the master file and all files in a volume scan to be present returns the masterfile if the volume scan can be processed.

### Parameters

**nowtime** [datetime object] the current time

**datacfg** [dict] dictionary containing the parameters to get the radar data

**last\_processed** [datetime or None] The end time of the previously processed radar volume

### Returns

**masterfile** [str or None] name of the master file. None if the volume was not complete

**masterdatatype\_descr** [str] the description of the master data type

**last\_processed** [datetime] True of all scans found

`pyrad.flow.flow_aux._wait_for_rainbow_datatypes(rainbow_files, period=30)`  
waits until the files for all rainbow data types are present.

### Parameters

**rainbow\_files** [list of strings] a list containing the names of all the rainbow files to wait for

**period** [int] the time it has to wait (s)

### Returns

**found\_all** [Boolean] True if all files were present. False otherwise

`pyrad.flow.flow_aux._warning_format` (*message*, *category*, *filename*, *lineno*, *file=None*,  
*line=None*)

`pyrad.flow.flow_aux.profiler` (*level=1*)

Function to be used as decorator for memory debugging. The function will be profiled or not according to its level respect to the global variable PROFILE\_LEVEL

**Parameters**

**level** [int] profiling level

**Returns**

**func or func wrapper** [function] The function or its wrapper for profiling



## PYRAD.FLOW.FLOW\_CONTROL

functions to control the Pyrad data processing flow

<code>main(cfgfile[, starttime, endtime, ...])</code>	Main flow control.
<code>main_rt(cfgfile_list[, starttime, endtime, ...])</code>	main flow control.
<code>main_cosmo(cfgfile[, starttime, endtime, ...])</code>	Main flow control.
<code>main_cosmo_rt(cfgfile_list[, starttime, ...])</code>	main flow control.

```
pyrad.flow.flow_control.main(cfgfile,      starttime=None,      endtime=None,      trajfile="",
                              trajtype='plane',      flashnr=0,      infostr="",      MULTI-
                              PROCESSING_DSET=False,      MULTIPROCESS-
                              ING_PROD=False,      PROFILE_MULTIPROCESSING=False,
                              USE_CHILD_PROCESS=False)
```

Main flow control. Processes radar data off-line over a period of time given either by the user, a trajectory file, or determined by the last volume processed and the current time. Multiple radars can be processed simultaneously

### Parameters

**cfgfile** [str] path of the main config file

**starttime, endtime** [datetime object] start and end time of the data to be processed

**trajfile** [str] path to file describing the trajectory

**trajtype** [str] type of trajectory file. Can be either 'plane', 'lightning' or 'proc\_periods'

**flashnr** [int] If larger than 0 will select a flash in a lightning trajectory file. If 0 the data corresponding to the trajectory of all flashes will be plotted

**infostr** [str] Information string about the actual data processing (e.g. 'RUN57'). This string is added to product files.

**MULTIPROCESSING\_DSET** [Bool] If true the generation of datasets at the same processing level will be parallelized

**MULTIPROCESSING\_PROD** [Bool] If true the generation of products from each dataset will be parallelized

**PROFILE\_MULTIPROCESSING** [Bool] If true and code parallelized the multiprocessing is profiled

**USE\_CHILD\_PROCESS** [Bool] If true the reading and processing of the data will be performed by a child process controlled by dask. This is done to make sure all memory used is released.

```
pyrad.flow.flow_control.main_cosmo(cfgfile, starttime=None, endtime=None, trajfile="", infostr="")
```

Main flow control. Processes radar data off-line over a period of time given either by the user, a trajectory file, or

determined by the last volume processed and the current time. Multiple radars can be processed simultaneously

#### Parameters

**cfgfile** [str] path of the main config file

**starttime, endtime** [datetime object] start and end time of the data to be processed

**trajfile** [str] path to file describing the trajectory

**infostr** [str] Information string about the actual data processing (e.g. 'RUN57'). This string is added to product files.

`pyrad.flow.flow_control.main_cosmo_rt(cfgfile_list, starttime=None, endtime=None, infostr_list=None, proc_period=60, proc_finish=None)`

main flow control. Processes radar data in real time. The start and end processing times can be determined by the user. This function is intended for a single radar

#### Parameters

**cfgfile\_list** [list of str] path of the main config files

**starttime, endtime** [datetime object] start and end time of the data to be processed

**infostr\_list** [list of str] Information string about the actual data processing (e.g. 'RUN57'). This string is added to product files.

**proc\_period** [int] period of time before starting a new processing round (seconds)

**proc\_finish** [int or None] if set to a value the program will be forced to shut down after the value (in seconds) from start time has been exceeded

#### Returns

**end\_proc** [Boolean] If true the program has ended successfully

`pyrad.flow.flow_control.main_rt(cfgfile_list, starttime=None, endtime=None, infostr_list=None, proc_period=60, proc_finish=None)`

main flow control. Processes radar data in real time. The start and end processing times can be determined by the user. This function is intended for a single radar

#### Parameters

**cfgfile\_list** [list of str] path of the main config files

**starttime, endtime** [datetime object] start and end time of the data to be processed

**infostr\_list** [list of str] Information string about the actual data processing (e.g. 'RUN57'). This string is added to product files.

**proc\_period** [int] period of time before starting a new processing round (seconds)

**proc\_finish** [int or None] if set to a value the program will be forced to shut down after the value (in seconds) from start time has been exceeded

#### Returns

**end\_proc** [Boolean] If true the program has ended successfully

## PYRAD.PROC.PROCESS\_AUX

Auxiliary functions. Functions to determine the process type, pass raw data to the product generation functions, save radar data and extract data at determined points or regions of interest.

<i>get_process_func</i> (dataset_type, dsname)	Maps the dataset type into its processing function and data set format associated.
<i>process_raw</i> (procstatus, dscfg[, radar_list])	Dummy function that returns the initial input data set
<i>process_save_radar</i> (procstatus, dscfg[, ...])	Dummy function that allows to save the entire radar object
<i>process_fixed_rng</i> (procstatus, dscfg[, ...])	Obtains radar data at a fixed range
<i>process_fixed_rng_span</i> (procstatus, dscfg[, ...])	For each azimuth-elevation gets the data within a fixed range span and computes a user-defined statistic: mean, min, max, mode, median
<i>process_roi</i> (procstatus, dscfg[, radar_list])	Obtains the radar data at a region of interest.
<i>process_azimuthal_average</i> (procstatus, dscfg)	Averages radar data in azimuth obtaining and RHI as a result
<i>process_radar_resampling</i> (procstatus, dscfg)	Resamples the radar data to mimic another radar with different geometry and antenna pattern
<i>_get_values_antenna_pattern</i> (radar, tadict, ...)	Get the values of a synthetic radar
<i>_create_target_radar</i> (radar, dscfg, ...[, ...])	Creates the target radar

```
pyrad.proc.process_aux._create_target_radar (radar,          dscfg,          fixed_angle_val,
                                              info,              field_names,
                                              change_antenna_pattern=False,  quantiles=[50])
```

Creates the target radar

### Parameters

- radar** [radar object] the radar object containing the observed data
- dscfg** [dict] dict with the configuration
- fixed\_angle\_val** [array of floats] array containing the fixed angles
- info** [str] String with info on the type of antenna
- field\_names** [list of str] the list of field names that the target radar will contain
- change\_antenna\_pattern** [bool] Whether the antenna pattern of the target radar is different from the observations radar
- quantiles** [list of floats] the quantiles to be computed if the target radar has a different antenna pattern

**Returns**

**target\_radar** [radar object] The target radar

`pyrad.proc.process_aux._get_values_antenna_pattern(radar, tadict, field_names)`

Get the values of a synthetic radar

**Parameters**

**radar** [radar object] The radar volume with the data

**tadict** [dict] A dictionary containing parameters useful for radar re-sampling

**field\_names** [list of str] list of names of the radar field

**Returns**

**target\_radar** [radar object] The synthetic radar

`pyrad.proc.process_aux.get_process_func(dataset_type, dsname)`

Maps the dataset type into its processing function and data set format associated.

**Parameters**

**dataset\_type** [str] The following is a list of data set types ordered by type of output dataset with the function they call. For details of what they do check the function documentation:

**‘VOL’ format output:** ‘ATTENUATION’: process\_attenuation ‘AZI\_AVG’: process\_azimuthal\_average ‘BIAS\_CORRECTION’: process\_correct\_bias ‘BIRDS\_ID’: process\_birds\_id ‘BIRD\_DENSITY’: process\_bird\_density ‘CCOR’: process\_ccor ‘CDF’: process\_cdf ‘CDR’: process\_cdr ‘CLT\_TO\_SAN’: process\_clt\_to\_echo\_id ‘COSMO’: process\_cosmo ‘COSMO\_LOOKUP’: process\_cosmo\_lookup\_table ‘DEM’: process\_dem ‘DEALIAS\_FOURDD’: process\_dealias\_fourdd ‘DEALIAS\_REGION’: process\_dealias\_region\_based ‘DEALIAS\_UNWRAP’: process\_dealias\_unwrap\_phase ‘DOPPLER\_VELOCITY’: process\_Doppler\_velocity ‘DOPPLER\_VELOCITY\_IQ’: process\_Doppler\_velocity\_iq ‘DOPPLER\_WIDTH’: process\_Doppler\_width ‘DOPPLER\_WIDTH\_IQ’: process\_Doppler\_width\_iq ‘ECHO\_FILTER’: process\_echo\_filter ‘FIELDS\_DIFF’: process\_fields\_diff ‘FIXED\_RNG’: process\_fixed\_rng ‘FIXED\_RNG\_SPAN’: process\_fixed\_rng\_span ‘HYDROCLASS’: process\_hydroclass ‘HGT’: process\_hgt ‘HGT\_LOOKUP’: process\_hgt\_lookup\_table ‘KDP\_LEASTSQUARE\_1W’: process\_kdp\_leastsquare\_single\_window ‘KDP\_LEASTSQUARE\_2W’: process\_kdp\_leastsquare\_double\_window ‘L’: process\_l ‘MEAN\_PHASE\_IQ’: process\_mean\_phase\_iq ‘NCVOL’: process\_save\_radar ‘NOISE\_POWER’: process\_noise\_power ‘OUTLIER\_FILTER’: process\_outlier\_filter ‘PhiDP’: process\_differential\_phase ‘PHIDP0\_CORRECTION’: process\_correct\_phidp0 ‘PHIDP0\_ESTIMATE’: process\_estimate\_phidp0 ‘PhiDP\_IQ’: process\_differential\_phase\_iq ‘PHIDP\_KDP\_KALMAN’: process\_phidp\_kdp\_Kalman ‘PHIDP\_KDP\_LP’: process\_phidp\_kdp\_lp ‘PHIDP\_KDP\_VULPIANI’: process\_phidp\_kdp\_Vulpiani ‘PHIDP\_SMOOTH\_1W’: process\_smooth\_phidp\_single\_window ‘PHIDP\_SMOOTH\_2W’: process\_smooth\_phidp\_double\_window ‘POL\_VARIABLES’: process\_pol\_variables ‘POL\_VARIABLES\_IQ’: process\_pol\_variables\_iq ‘PWR’: process\_signal\_power ‘RADAR\_RESAMPLING’: process\_radar\_resampling ‘RAINRATE’: process\_rainrate ‘RAW’: process\_raw ‘REFLECTIVITY’: process\_reflectivity ‘REFLECTIVITY\_IQ’: process\_reflectivity\_iq ‘RCS’: process\_rcs ‘RCS\_PR’: process\_rcs\_pr ‘RhoHV’: process\_rhohv ‘RhoHV\_IQ’: process\_rhohv\_iq ‘RHOHV\_CORRECTION’: process\_correct\_noise\_rhohv ‘RHOHV\_RAIN’: process\_rhohv\_rain ‘ROI’:



process\_roi 'SAN': process\_echo\_id 'SELFCONSISTENCY\_BIAS': process\_selfconsistency\_bias 'SELFCONSISTENCY\_KDP\_PHIDP': process\_selfconsistency\_kdp\_phidp 'SNR': process\_snr 'SNR\_FILTER': process\_filter\_snr 'ST1\_IQ': process\_st1\_iq 'ST2\_IQ': process\_st2\_iq 'TRAJ\_TRT': process\_traj\_trt 'TRAJ\_TRT\_CONTOUR': process\_traj\_trt\_contour 'TURBULENCE': process\_turbulence 'VAD': process\_vad 'VEL\_FILTER': process\_filter\_vel\_diff 'VIS': process\_visibility 'VIS\_FILTER': process\_filter\_visibility 'VOL\_REFL': process\_vol\_refl 'WBN': process\_wbn\_iq 'WIND\_VEL': process\_wind\_vel 'WINDSHEAR': process\_windshear 'ZDR': process\_differential\_reflectivity 'ZDR\_IQ': process\_differential\_reflectivity\_iq 'ZDR\_PREC': process\_zdr\_precip 'ZDR\_SNOW': process\_zdr\_snow

**'SPECTRA' format output:** 'FFT': process\_fft 'FILTER\_0DOPPLER': process\_filter\_0Doppler 'FILTER\_SPECTRA\_NOISE': process\_filter\_spectra\_noise 'IFFT': process\_ifft 'RAW\_IQ': process\_raw\_iq 'RAW\_SPECTRA': process\_raw\_spectra 'SPECTRA\_ANGULAR\_AVERAGE': process\_spectra\_ang\_avg 'SPECTRA\_POINT': process\_spectra\_point 'SPECTRAL\_NOISE': process\_spectral\_noise 'SPECTRAL\_PHASE': process\_spectral\_phase 'SPECTRAL\_POWER': process\_spectral\_power 'SPECTRAL\_REFLECTIVITY': process\_spectral\_reflectivity 'sPhiDP': process\_spectral\_differential\_phase 'sRhoHV': process\_spectral\_RhoHV 'SRHOHV\_FILTER': process\_filter\_srhoHV 'sZDR': process\_spectral\_differential\_reflectivity

**'COLOCATED\_GATES' format output:** 'COLOCATED\_GATES': process\_collocated\_gates

**'COSMO\_COORD' format output:** 'COSMO\_COORD': process\_cosmo\_coord 'HGT\_COORD': process\_hgt\_coord

**'COSMO2RADAR' format output:** 'COSMO2RADAR': process\_cosmo\_to\_radar

**'GRID' format output:** 'RAW\_GRID': process\_raw\_grid 'GRID': process\_grid

**'GRID\_TIMEAVG' format output:** 'GRID\_TIME\_STATS': process\_grid\_time\_stats 'GRID\_TIME\_STATS2': process\_grid\_time\_stats2

**'INTERCOMP' format output:** 'INTERCOMP': process\_intercomp 'INTERCOMP\_FIELDS': process\_intercomp\_fields 'INTERCOMP\_TIME\_AVG': process\_intercomp\_time\_avg

**'ML' format output:** 'ML\_DETECTION': process\_melting\_layer

**'MONITORING' format output:** 'GC\_MONITORING': process\_gc\_monitoring 'MONITORING': process\_monitoring

**'OCCURRENCE' format output:** 'OCCURRENCE': process\_occurrence 'OCCURRENCE\_PERIOD': process\_occurrence\_period 'TIMEAVG\_STD': process\_time\_avg\_std

**'QVP' format output:** 'EVP': process\_evp 'QVP': process\_qvp 'rQVP': process\_rqvp 'SVP': process\_svp 'TIME\_HEIGHT': process\_time\_height

**'SPARSE\_GRID' format output:** 'ZDR\_COLUMN': process\_zdr\_column

**'SUN\_HITS' format output:** 'SUN\_HITS': process\_sun\_hits

**'TIMEAVG' format output:** 'FLAG\_TIME\_AVG': process\_time\_avg\_flag 'TIME\_AVG': process\_time\_avg 'WEIGHTED\_TIME\_AVG': process\_weighted\_time\_avg 'TIME\_STATS': process\_time\_stats 'TIME\_STATS2': process\_time\_stats2 'RAIN\_ACCU': process\_rainfall\_accumulation

**‘TIMESERIES’ format output:** ‘GRID\_POINT\_MEASUREMENT’: process\_grid\_point  
‘POINT\_MEASUREMENT’: process\_point\_measurement  
‘TRAJ\_ANTENNA\_PATTERN’: process\_traj\_antenna\_pattern  
‘TRAJ\_ATPLANE’: process\_traj\_atplane  
‘TRAJ\_LIGHTNING’: process\_traj\_lightning

**‘TRAJ\_ONLY’ format output:** ‘TRAJ’: process\_trajectory

**dsname** [str] Name of dataset

### Returns

**func\_name** [str or processing function] pyrad function used to process the data set type

**dsformat** [str] data set format, i.e.: ‘VOL’, etc.

`pyrad.proc.process_aux.process_azimuthal_average(procstatus, dscfg, radar_list=None)`  
Averages radar data in azimuth obtaining and RHI as a result

### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The data type where we want to extract the point measurement

**angle** [float or None. Dataset keyword] The

delta\_azi : float. Dataset keyword

avg\_type : str. Dataset keyword

**nvalid\_min** [int. Dataset keyword] the (minimum) radius of the region of interest in m.  
Default half the largest resolution

**radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

**new\_dataset** [dict] dictionary containing the gridded data

**ind\_rad** [int] radar index

`pyrad.proc.process_aux.process_fixed_rng(procstatus, dscfg, radar_list=None)`  
Obtains radar data at a fixed range

### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of strings. Dataset keyword] The fields we want to extract

**rng** [float. Dataset keyword] The fixed range [m]

**RngTol** [float. Dataset keyword] The tolerance between the nominal range and the radar range

**ele\_min, ele\_max, azi\_min, azi\_max** [floats. Dataset keyword] The azimuth and elevation limits of the data [deg]

**radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

**new\_dataset** [dict] dictionary containing the data and metadata at the point of interest

**ind\_rad** [int] radar index

`pyrad.proc.process_aux.process_fixed_rng_span(procstatus, dscfg, radar_list=None)`

For each azimuth-elevation gets the data within a fixed range span and computes a user-defined statistic: mean, min, max, mode, median

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of strings. Dataset keyword] The fields we want to extract

**rmin, rmax** [float. Dataset keyword] The range limits [m]

**ele\_min, ele\_max, azi\_min, azi\_max** [floats. Dataset keyword] The azimuth and elevation limits of the data [deg]

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the data and metadata at the point of interest

**ind\_rad** [int] radar index

`pyrad.proc.process_aux.process_radar_resampling(procstatus, dscfg, radar_list=None)`

Resamples the radar data to mimic another radar with different geometry and antenna pattern

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries]

**datatype** [list of string. Dataset keyword] The input data types

**antennaType** [str. Dataset keyword] Type of antenna of the radar we want to get the view from. Can be AZIMUTH, ELEVATION, LOWBEAM, HIGHBEAM

**par\_azimuth\_antenna** [dict. Global keyword] Dictionary containing the parameters of the PAR azimuth antenna, i.e. name of the file with the antenna elevation pattern and fixed antenna angle

**par\_elevation\_antenna** [dict. Global keyword] Dictionary containing the parameters of the PAR elevation antenna, i.e. name of the file with the antenna azimuth pattern and fixed antenna angle

**asr\_lowbeam\_antenna** [dict. Global keyword] Dictionary containing the parameters of the ASR low beam antenna, i.e. name of the file with the antenna elevation pattern and fixed antenna angle

**asr\_highbeam\_antenna** [dict. Global keyword] Dictionary containing the parameters of the ASR high beam antenna, i.e. name of the file with the antenna elevation pattern and fixed antenna angle

**target\_radar\_pos** [dict. Global keyword] Dictionary containing the latitude, longitude and altitude of the radar we want to get the view from. If not specifying it will assume the radar is collocated

**change\_antenna\_pattern** [Bool. Dataset keyword] If true the target radar has a different antenna pattern than the observations radar

**rhi\_resolution** [Bool. Dataset keyword] Resolution of the synthetic RHI used to compute the data as viewed from the synthetic radar [deg]. Default 0.5

**max\_altitude** [float. Dataset keyword] Max altitude of the data to use when computing the view from the synthetic radar [m MSL]. Default 12000.

**latlon\_tol** [float. Dataset keyword] The tolerance in latitude and longitude to determine which synthetic radar gates are co-located with real radar gates [deg]. Default 0.04

**alt\_tol** [float. Dataset keyword] The tolerance in altitude to determine which synthetic radar gates are co-located with real radar gates [m]. Default 1000.

**pattern\_thres** [float. Dataset keyword] The minimum of the sum of the weights given to each value in order to consider the weighted quantile valid. It is related to the number of valid data points

**data\_is\_log** [dict. Dataset keyword] Dictionary specifying for each field if it is in log (True) or linear units (False). Default False

**use\_nans** [dict. Dataset keyword] Dictionary specifying whether the nans have to be used in the computation of the statistics for each field. Default False

**nan\_value** [dict. Dataset keyword] Dictionary with the value to use to substitute the NaN values when computing the statistics of each field. Default 0

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [Trajectory object] dictionary containing the new radar

**ind\_rad** [int] radar index

`pyrad.proc.process_aux.process_raw(procstatus, dscfg, radar_list=None)`

Dummy function that returns the initial input data set

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_aux.process_roi(procstatus, dscfg, radar_list=None)`

Obtains the radar data at a region of interest.

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The data type where we want to extract the point measurement

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the data and metadata at the point of interest

**ind\_rad** [int] radar index

`pyrad.proc.process_aux.process_save_radar` (*procstatus*, *dscfg*, *radar\_list=None*)

Dummy function that allows to save the entire radar object

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index



## PYRAD.PROC.PROCESS\_CALIB

Functions for monitoring data quality and correct bias and noise effects

<code>process_correct_bias(procstatus, dscfg[, ...])</code>	Corrects a bias on the data
<code>process_correct_noise_rho_hv(procstatus, dscfg)</code>	identifies echoes as 0: No data, 1: Noise, 2: Clutter, 3: Precipitation
<code>process_gc_monitoring(procstatus, dscfg[, ...])</code>	computes ground clutter monitoring statistics
<code>process_occurrence(procstatus, dscfg[, ...])</code>	computes the frequency of occurrence of data.
<code>process_time_avg_std(procstatus, dscfg[, ...])</code>	computes the average and standard deviation of data.
<code>process_occurrence_period(procstatus, dscfg)</code>	computes the frequency of occurrence over a long period of time by adding together shorter periods
<code>process_sun_hits(procstatus, dscfg[, radar_list])</code>	monitoring of the radar using sun hits

`pyrad.proc.process_calib.process_correct_bias (procstatus, dscfg, radar_list=None)`  
Corrects a bias on the data

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
- datatype** [string. Dataset keyword] The data type to correct for bias
  - bias** [float. Dataset keyword] The bias to be corrected [dB]. Default 0
- radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

- new\_dataset** [dict] dictionary containing the output
- ind\_rad** [int] radar index

`pyrad.proc.process_calib.process_correct_noise_rho_hv (procstatus, dscfg, radar_list=None)`  
identifies echoes as 0: No data, 1: Noise, 2: Clutter, 3: Precipitation

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
- datatype** [list of string. Dataset keyword] The data types used in the correction
- radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_calib.process_gc_monitoring(procstatus, dscfg, radar_list=None)`  
computes ground clutter monitoring statistics

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**excessgatespath** [str. Config keyword] The path to the gates in excess of quantile location

**excessgates\_fname** [str. Dataset keyword] The name of the gates in excess of quantile file

**datatype** [list of string. Dataset keyword] The input data types

**step** [float. Dataset keyword] The width of the histogram bin. Default is None. In that case the default step in function `get_histogram_bins` is used

**regular\_grid** [Boolean. Dataset keyword] Whether the radar has a Boolean grid or not.  
Default False

**val\_min** [Float. Dataset keyword] Minimum value to consider that the gate has signal.  
Default None

**filter\_prec** [str. Dataset keyword] Give which type of volume should be filtered. None, no filtering; keep\_wet, keep wet volumes; keep\_dry, keep dry volumes.

**rmax\_prec** [float. Dataset keyword] Maximum range to consider when looking for wet gates [m]

**percent\_prec\_max** [float. Dataset keyword] Maxim percentage of wet gates to consider the volume dry

**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [Radar] radar object containing histogram data

**ind\_rad** [int] radar index

`pyrad.proc.process_calib.process_occurrence(procstatus, dscfg, radar_list=None)`  
computes the frequency of occurrence of data. It looks only for gates where data is present.

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**regular\_grid** [Boolean. Dataset keyword] Whether the radar has a Boolean grid or not.  
Default False

**rmin, rmax** [float. Dataset keyword] minimum and maximum ranges where the computation takes place. If -1 the whole range is considered. Default is -1

**val\_min** [Float. Dataset keyword] Minimum value to consider that the gate has signal.  
Default None



**filter\_prec** [str. Dataset keyword] Give which type of volume should be filtered. None, no filtering; keep\_wet, keep wet volumes; keep\_dry, keep dry volumes.

**rmax\_prec** [float. Dataset keyword] Maximum range to consider when looking for wet gates [m]

**percent\_prec\_max** [float. Dataset keyword] Maxim percentage of wet gates to consider the volume dry

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_calib.process_occurrence_period` (*procstatus*, *dscfg*,  
*radar\_list=None*)  
computes the frequency of occurrence over a long period of time by adding together shorter periods

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**regular\_grid** [Boolean. Dataset keyword] Whether the radar has a Boolean grid or not.  
Default False

**rmin, rmax** [float. Dataset keyword] minimum and maximum ranges where the computation takes place. If -1 the whole range is considered. Default is -1

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_calib.process_sun_hits` (*procstatus*, *dscfg*, *radar\_list=None*)  
monitoring of the radar using sun hits

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**rmin** [float. Dataset keyword] minimum range where to look for a sun hit signal [m].  
Default 50000.

**hmin** [float. Dataset keyword] minimum altitude where to look for a sun hit signal [m MSL]. Default 10000. The actual range from which a sun hit signal will be search will be the minimum between rmin and the range from which the altitude is higher than hmin.

**delev\_max** [float. Dataset keyword] maximum elevation distance from nominal radar elevation where to look for a sun hit signal [deg]. Default 1.5

**dazim\_max** [float. Dataset keyword] maximum azimuth distance from nominal radar elevation where to look for a sun hit signal [deg]. Default 1.5

**elmin** [float. Dataset keyword] minimum radar elevation where to look for sun hits [deg]. Default 1.

**nbins\_min** [int. Dataset keyword.] minimum number of range bins that have to contain signal to consider the ray a potential sun hit. Default 10.

**attg** [float. Dataset keyword] gaseous attenuation. Default None

**max\_std\_pwr** [float. Dataset keyword] maximum standard deviation of the signal power to consider the data a sun hit [dB]. Default 2.

**max\_std\_zdr** [float. Dataset keyword] maximum standard deviation of the ZDR to consider the data a sun hit [dB]. Default 2.

**az\_width\_co** [float. Dataset keyword] co-polar antenna azimuth width (convoluted with sun width) [deg]. Default None

**el\_width\_co** [float. Dataset keyword] co-polar antenna elevation width (convoluted with sun width) [deg]. Default None

**az\_width\_cross** [float. Dataset keyword] cross-polar antenna azimuth width (convoluted with sun width) [deg]. Default None

**el\_width\_cross** [float. Dataset keyword] cross-polar antenna elevation width (convoluted with sun width) [deg]. Default None

**ndays** [int. Dataset keyword] number of days used in sun retrieval. Default 1

**coeff\_band** [float. Dataset keyword] multiply coefficient to transform pulse width into receiver bandwidth

**frequency** [float. Dataset keyword] the radar frequency [Hz]. If None that of the key frequency in attribute instrument\_parameters of the radar object will be used. If the key or the attribute are not present frequency dependent parameters will not be computed

**beamwidth** [float. Dataset keyword] the antenna beamwidth [deg]. If None that of the keys radar\_beam\_width\_h or radar\_beam\_width\_v in attribute instrument\_parameters of the radar object will be used. If the key or the attribute are not present the beamwidth dependent parameters will not be computed

**pulse\_width** [float. Dataset keyword] the pulse width [s]. If None that of the key pulse\_width in attribute instrument\_parameters of the radar object will be used. If the key or the attribute are not present the pulse width dependent parameters will not be computed

**ray\_angle\_res** [float. Dataset keyword] the ray angle resolution [deg]. If None that of the key ray\_angle\_res in attribute instrument\_parameters of the radar object will be used. If the key or the attribute are not present the ray angle resolution parameters will not be computed

**AntennaGainH, AntennaGainV** [float. Dataset keyword] the horizontal (vertical) polarization antenna gain [dB]. If None that of the attribute instrument\_parameters of the radar object will be used. If the key or the attribute are not present the ray angle resolution parameters will not be computed

**radar\_list** [list of Radar objects] Optional. list of radar objects

## Returns

**sun\_hits\_dict** [dict] dictionary containing a radar object, a sun\_hits dict and a sun\_retrieval dictionary

**ind\_rad** [int] radar index

`pyrad.proc.process_calib.process_time_avg_std(procstatus, dscfg, radar_list=None)`  
computes the average and standard deviation of data. It looks only for gates where data is present.

#### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
- datatype** [list of string. Dataset keyword] The input data types
  - regular\_grid** [Boolean. Dataset keyword] Whether the radar has a Boolean grid or not.  
Default False
  - rmin, rmax** [float. Dataset keyword] minimum and maximum ranges where the computation takes place. If -1 the whole range is considered. Default is -1
  - val\_min** [Float. Dataset keyword] Minimum reflectivity value to consider that the gate has signal. Default None
  - filter\_prec** [str. Dataset keyword] Give which type of volume should be filtered. None, no filtering; keep\_wet, keep wet volumes; keep\_dry, keep dry volumes.
  - rmax\_prec** [float. Dataset keyword] Maximum range to consider when looking for wet gates [m]
  - percent\_prec\_max** [float. Dataset keyword] Maxim percentage of wet gates to consider the volume dry
  - lin\_trans** [Boolean. Dataset keyword] If True the data will be transformed into linear units.  
Default False
- radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

- new\_dataset** [dict] dictionary containing the output
- ind\_rad** [int] radar index



## PYRAD.PROC.PROCESS\_COSMO

Functions to manage COSMO data

<i>process_cosmo</i> (procstatus, dscfg[, radar_list])	Gets COSMO data and put it in radar coordinates
<i>process_hzt</i> (procstatus, dscfg[, radar_list])	Gets iso0 degree data in HZT format and put it in radar coordinates
<i>process_cosmo_lookup_table</i> (procstatus, dscfg)	Gets COSMO data and put it in radar coordinates using look up tables computed or loaded when initializing
<i>process_hzt_lookup_table</i> (procstatus, dscfg)	Gets HZT data and put it in radar coordinates using look up tables computed or loaded when initializing
<i>process_cosmo_to_radar</i> (procstatus, dscfg[, ...])	Gets COSMO data and put it in radar coordinates using look up tables
<i>process_cosmo_coord</i> (procstatus, dscfg[, ...])	Gets the COSMO indices corresponding to each cosmo coordinates
<i>process_hzt_coord</i> (procstatus, dscfg[, ...])	Gets the HZT indices corresponding to each HZT coordinates

`pyrad.proc.process_cosmo.process_cosmo (procstatus, dscfg, radar_list=None)`

Gets COSMO data and put it in radar coordinates

### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] arbitrary data type

**keep\_in\_memory** [int. Dataset keyword] if set keeps the COSMO data dict, the COSMO coordinates dict and the COSMO field in radar coordinates in memory

**regular\_grid** [int. Dataset keyword] if set it is assume that the radar has a grid constant in time and there is no need to compute a new COSMO field if the COSMO data has not changed

**cosmo\_type** [str. Dataset keyword] name of the COSMO field to process. Default TEMP

**cosmo\_variables** [list of strings. Dataset keyword] Py-art name of the COSMO fields. Default temperature

**radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_cosmo.process_cosmo_coord(procstatus, dscfg, radar_list=None)`

Gets the COSMO indices corresponding to each cosmo coordinates

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] arbitrary data type

**cosmopath** [string. General keyword] path where to store the look up table

**model** [string. Dataset keyword] The COSMO model to use. Can be cosmo-1, cosmo-2, cosmo-7

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_cosmo.process_cosmo_lookup_table(procstatus, dscfg, radar_list=None)`

Gets COSMO data and put it in radar coordinates using look up tables computed or loaded when initializing

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] arbitrary data type

**lookup\_table** [int. Dataset keyword] if set a pre-computed look up table for the COSMO coordinates is loaded. Otherwise the look up table is computed taking the first radar object as reference

**regular\_grid** [int. Dataset keyword] if set it is assume that the radar has a grid constant in time and therefore there is no need to interpolate the COSMO field in memory to the current radar grid

**cosmo\_type** [str. Dataset keyword] name of the COSMO field to process. Default TEMP

**cosmo\_variables** [list of strings. Dataset keyword] Py-art name of the COSMO fields. Default temperature

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_cosmo.process_cosmo_to_radar(procstatus, dscfg, radar_list=None)`

Gets COSMO data and put it in radar coordinates using look up tables

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] arbitrary data type

**cosmo\_type** [str. Dataset keyword] name of the COSMO field to process. Default TEMP

**cosmo\_variables** [list of strings. Dataset keyword] Py-art name of the COSMO fields.  
Default temperature

**cosmo\_time\_index\_min, cosmo\_time\_index\_max** [int] minimum and maximum indices  
of the COSMO data to retrieve. If a value is provided only data corresponding to the time  
indices within the interval will be used. If None all data will be used. Default None

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_cosmo.process_hzt(procstatus, dscfg, radar_list=None)`

Gets iso0 degree data in HZT format and put it in radar coordinates

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**metranet\_read\_lib** [str. Global keyword] Type of METRANET reader library used to read  
the data. Can be 'C' or 'python'

**datatype** [string. Dataset keyword] arbitrary data type

**keep\_in\_memory** [int. Dataset keyword] if set keeps the COSMO data dict, the COSMO  
coordinates dict and the COSMO field in radar coordinates in memory

**regular\_grid** [int. Dataset keyword] if set it is assume that the radar has a grid constant in  
time and there is no need to compute a new COSMO field if the COSMO data has not  
changed

**cosmo\_type** [str. Dataset keyword] name of the COSMO field to process. Default TEMP

**cosmo\_variables** [list of strings. Dataset keyword] Py-art name of the COSMO fields.  
Default temperature

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_cosmo.process_hzt_coord(procstatus, dscfg, radar_list=None)`

Gets the HZT indices corresponding to each HZT coordinates

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**metranet\_read\_lib** [str. Global keyword] Type of METRANET reader library used to read  
the data. Can be 'C' or 'python'

**datatype** [string. Dataset keyword] arbitrary data type

**cosmopath** [string. General keyword] path where to store the look up table

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_cosmo.process_hzt_lookup_table` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Gets HZT data and put it in radar coordinates using look up tables computed or loaded when initializing

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**metranet\_read\_lib** [str. Global keyword] Type of METRANET reader library used to read the data. Can be 'C' or 'python'

**datatype** [string. Dataset keyword] arbitrary data type

**lookup\_table** [int. Dataset keyword] if set a pre-computed look up table for the COSMO coordinates is loaded. Otherwise the look up table is computed taking the first radar object as reference

**regular\_grid** [int. Dataset keyword] if set it is assume that the radar has a grid constant in time and therefore there is no need to interpolate the COSMO field in memory to the current radar grid

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index



## PYRAD.PROC.PROCESS\_DEM

Functions to manage DEM data

<i>process_dem</i> (procstatus, dscfg[, radar_list])	Gets COSMO data and put it in radar coordinates
<i>process_visibility</i> (procstatus, dscfg[, ...])	Gets the visibility in percentage from the minimum visible elevation.

---

`pyrad.proc.process_dem.process_dem(procstatus, dscfg, radar_list=None)`  
Gets COSMO data and put it in radar coordinates

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
- datatype** [string. Dataset keyword] arbitrary data type
  - keep\_in\_memory** [int. Dataset keyword] if set keeps the COSMO data dict, the COSMO coordinates dict and the COSMO field in radar coordinates in memory. Default False
  - regular\_grid** [int. Dataset keyword] if set it is assume that the radar has a grid constant in time and there is no need to compute a new COSMO field if the COSMO data has not changed. Default False
  - dem\_field** [str. Dataset keyword] name of the DEM field to process
  - demfile** [str. Dataset keyword] Name of the file containing the DEM data
- radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

- new\_dataset** [dict] dictionary containing the output
- ind\_rad** [int] radar index

`pyrad.proc.process_dem.process_visibility(procstatus, dscfg, radar_list=None)`  
Gets the visibility in percentage from the minimum visible elevation. Anything with elevation lower than the minimum visible elevation plus and offset is set to 0 while above is set to 100.

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
- datatype** [string. Dataset keyword] arbitrary data type
  - offset** [float. Dataset keyword] The offset above the minimum visibility that must be filtered

**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

## PYRAD.PROC.PROCESS\_DOPPLER

Functions for processing Doppler related parameters

<code>process_turbulence(procstatus, dscfg[, ...])</code>	Computes turbulence from the Doppler spectrum width and reflectivity using the PyTDA package
<code>process_dealias_fourdd(procstatus, dscfg[, ...])</code>	Dealiases the Doppler velocity field using the 4DD technique from Curtis and Houze, 2001
<code>process_dealias_region_based(procstatus, dscfg)</code>	Dealiases the Doppler velocity field using a region based algorithm
<code>process_dealias_unwrap_phase(procstatus, dscfg)</code>	Dealiases the Doppler velocity field using multi-dimensional phase unwrapping
<code>process_wind_vel(procstatus, dscfg[, radar_list])</code>	Estimates the horizontal or vertical component of the wind from the radial velocity
<code>process_windshear(procstatus, dscfg[, ...])</code>	Estimates the wind shear from the wind velocity
<code>process_vad(procstatus, dscfg[, radar_list])</code>	Estimates vertical wind profile using the VAD (velocity Azimuth Display) technique

`pyrad.proc.process_Doppler.process_dealias_fourdd` (*procstatus*, *dscfg*,  
*radar\_list=None*)  
Dealiases the Doppler velocity field using the 4DD technique from Curtis and Houze, 2001

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
  - datatype** [string. Dataset keyword] The input data type
  - filt** [int. Dataset keyword] Flag controlling Bergen and Albers filter, 1 = yes, 0 = no.
  - sign** [int. Dataset keyword] Sign convention which the radial velocities in the volume created from the sounding data will will. This should match the convention used in the radar data. A value of 1 represents when positive values velocities are towards the radar, -1 represents when negative velocities are towards the radar.
- radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

- new\_dataset** [dict] dictionary containing the output
- ind\_rad** [int] radar index

`pyrad.proc.process_Doppler.process_dealias_region_based` (*procstatus*, *dscfg*,  
*radar\_list=None*)  
Dealiases the Doppler velocity field using a region based algorithm

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The input data type

**interval\_splits** [int, optional] Number of segments to split the nyquist interval into when finding regions of similar velocity. More splits creates a larger number of initial regions which takes longer to process but may result in better dealiasing. The default value of 3 seems to be a good compromise between performance and artifact free dealiasing. This value is not used if the `interval_limits` parameter is not None.

**skip\_between\_rays, skip\_along\_ray** [int, optional] Maximum number of filtered gates to skip over when joining regions, gaps between region larger than this will not be connected. Parameters specify the maximum number of filtered gates between and along a ray. Set these parameters to 0 to disable unfolding across filtered gates.

**centered** [bool, optional] True to apply centering to each sweep after the dealiasing algorithm so that the average number of unfolding is near 0. False does not apply centering which may results in individual sweeps under or over folded by the nyquist interval.

**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_Doppler.process_dealias_unwrap_phase` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Dealiases the Doppler velocity field using multi-dimensional phase unwrapping

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The input data type

**unwrap\_unit** [{ 'ray', 'sweep', 'volume' }, optional] Unit to unwrap independently. 'ray' will unwrap each ray individually, 'sweep' each sweep, and 'volume' will unwrap the entire volume in a single pass. 'sweep', the default, often gives superior results when the lower sweeps of the radar volume are contaminated by clutter. 'ray' does not use the `gatefilter` parameter and rays where gates are masked will result in poor dealiasing for that ray.

**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_Doppler.process_turbulence` (*procstatus*, *dscfg*, *radar\_list=None*)

Computes turbulence from the Doppler spectrum width and reflectivity using the PyTDA package

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The input data type

**radius** [float. Dataset keyword] Search radius for calculating Eddy Dissipation Rate (EDR). Default 2

**split\_cut** [Bool. Dataset keyword] Set to True for split-cut volumes. Default False

**max\_split\_cut** [Int. Dataset keyword] Total number of tilts that are affected by split cuts. Only relevant if split\_cut=True. Default 2

**xran, yran** [float array. Dataset keyword] Spatial range in X,Y to consider. Default [-100, 100] for both X and Y

**beamwidth** [Float. Dataset keyword] Radar beamwidth. Default None. If None it will be obtained from the radar object metadata. If cannot be obtained defaults to 1 deg.

**compute\_gate\_pos** [Bool. Dataset keyword] If True the gate position is going to be computed in PyTDA. Otherwise the position from the radar object is used. Default False

**verbose** [Bool. Dataset keyword] True for verbose output. Default False

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_Doppler.process_vad(procstatus, dscfg, radar_list=None)`  
Estimates vertical wind profile using the VAD (velocity Azimuth Display) technique

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The input data type

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_Doppler.process_wind_vel(procstatus, dscfg, radar_list=None)`  
Estimates the horizontal or vertical component of the wind from the radial velocity

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The input data type

**vert\_proj** [Boolean] If true the vertical projection is computed. Otherwise the horizontal projection is computed

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_Doppler.process_windshear` (*procstatus, dscfg, radar\_list=None*)

Estimates the wind shear from the wind velocity

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The input data type

**az\_tol** [float] The tolerance in azimuth when looking for gates on top of the gate when computation is performed

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

## PYRAD.PROC.PROCESS\_ECHOCCLASS

Functions for echo classification and filtering

<i>process_echo_id</i> (procstatus, dscfg[, radar_list])	identifies echoes as 0: No data, 1: Noise, 2: Clutter, 3: Precipitation
<i>process_birds_id</i> (procstatus, dscfg[, radar_list])	identifies echoes as 0: No data, 1: Noise, 2: Clutter, 3: Birds
<i>process_clt_to_echo_id</i> (procstatus, dscfg[, ...])	Converts clutter exit code from rad4alp into pyrad echo ID
<i>process_echo_filter</i> (procstatus, dscfg[, ...])	Masks all echo types that are not of the class specified in keyword <i>echo_type</i>
<i>process_cdf</i> (procstatus, dscfg[, radar_list])	Collects the fields necessary to compute the Cumulative Distribution Function
<i>process_filter_snr</i> (procstatus, dscfg[, ...])	filters out low SNR echoes
<i>process_filter_vel_diff</i> (procstatus, dscfg[, ...])	filters out range gates that could not be used for Doppler velocity estimation
<i>process_filter_visibility</i> (procstatus, dscfg)	filters out rays gates with low visibility and corrects the reflectivity
<i>process_outlier_filter</i> (procstatus, dscfg[, ...])	filters out gates which are outliers respect to the surrounding
<i>process_hydroclass</i> (procstatus, dscfg[, ...])	Classifies precipitation echoes
<i>process_melting_layer</i> (procstatus, dscfg[, ...])	Detects the melting layer
<i>process_zdr_column</i> (procstatus, dscfg[, ...])	Detects ZDR columns

`pyrad.proc.process_echoclass.process_birds_id` (*procstatus*, *dscfg*, *radar\_list=None*)  
 identifies echoes as 0: No data, 1: Noise, 2: Clutter, 3: Birds

### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
     **datatype** [list of string. Dataset keyword] The input data types  
**radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_cdf` (*procstatus*, *dscfg*, *radar\_list=None*)  
 Collects the fields necessary to compute the Cumulative Distribution Function

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_clt_to_echo_id(procstatus, dscfg, radar_list=None)`

Converts clutter exit code from rad4alp into pyrad echo ID

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_echo_filter(procstatus, dscfg, radar_list=None)`  
Masks all echo types that are not of the class specified in keyword echo\_type

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
    **echo\_type** [int or list of ints] The type of echoes to keep: 1 noise, 2 clutter, 3 precipitation.  
    Default 3  
**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_echo_id(procstatus, dscfg, radar_list=None)`  
identifies echoes as 0: No data, 1: Noise, 2: Clutter, 3: Precipitation

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
**radar\_list** [list of Radar objects] Optional. list of radar objects



**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_filter_snr` (*procstatus*, *dscfg*, *radar\_list=None*)  
filters out low SNR echoes

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**SNRmin** [float. Dataset keyword] The minimum SNR to keep the data.

**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_filter_vel_diff` (*procstatus*, *dscfg*,  
*radar\_list=None*)  
filters out range gates that could not be used for Doppler velocity estimation

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_filter_visibility` (*procstatus*, *dscfg*,  
*radar\_list=None*)  
filters out rays gates with low visibility and corrects the reflectivity

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**VISmin** [float. Dataset keyword] The minimum visibility to keep the data.

**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_hydroclass` (*procstatus, dscfg, radar\_list=None*)  
Classifies precipitation echoes

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

- datatype** [list of string. Dataset keyword] The input data types
- HYDRO\_METHOD** [string. Dataset keyword] The hydrometeor classification method. One of the following: SEMISUPERVISED
- RADARCENTROIDS** [string. Dataset keyword] Used with HYDRO\_METHOD SEMISUPERVISED. The name of the radar of which the derived centroids will be used. One of the following: A Albis, L Lema, P Plaine Morte, DX50
- compute\_entropy** [bool. Dataset keyword] If true the entropy is computed and the field `hydroclass_entropy` is output
- output\_distances** [bool. Dataset keyword] If true the de-mixing algorithm based on the distances to the centroids is computed and the field proportions of each hydrometeor in the radar range gate is output
- vectorize** [bool. Dataset keyword] If true a vectorized version of the algorithm is used

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_melting_layer` (*procstatus, dscfg, radar\_list=None*)

Detects the melting layer

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

- datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_outlier_filter` (*procstatus, dscfg, radar\_list=None*)

filters out gates which are outliers respect to the surrounding

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

- datatype** [list of string. Dataset keyword] The input data types

**threshold** [float. Dataset keyword] The distance between the value of the examined range gate and the median of the surrounding gates to consider the gate an outlier

**nb** [int. Dataset keyword] The number of neighbours (to one side) to analyse. i.e. 2 would correspond to 24 gates

**nb\_min** [int. Dataset keyword] Minimum number of neighbouring gates to consider the examined gate valid

**percentile\_min, percentile\_max** [float. Dataset keyword] gates below (above) these percentiles (computed over the sweep) are considered potential outliers and further examined

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_echoclass.process_zdr_column(procstatus, dscfg, radar_list=None)`  
Detects ZDR columns

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index



## PYRAD.PROC.PROCESS\_GRID

Functions to processes gridded data.

<i>process_raw_grid</i> (procstatus, dscfg[, radar_list])	Dummy function that returns the initial input data set
<i>process_grid</i> (procstatus, dscfg[, radar_list])	Puts the radar data in a regular grid
<i>process_grid_point</i> (procstatus, dscfg[, ...])	Obtains the grid data at a point location.
<i>process_grid_time_stats</i> (procstatus, dscfg[, ...])	computes the temporal statistics of a field
<i>process_grid_time_stats2</i> (procstatus, dscfg)	computes the temporal mean of a field

`pyrad.proc.process_grid.process_grid(procstatus, dscfg, radar_list=None)`  
Puts the radar data in a regular grid

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
- datatype** [string. Dataset keyword] The data type where we want to extract the point measurement
  - gridconfig** [dictionary. Dataset keyword] Dictionary containing some or all of this keywords: xmin, xmax, ymin, ymax, zmin, zmax : floats
    - minimum and maximum horizontal distance from grid origin [km] and minimum and maximum vertical distance from grid origin [m] Defaults -40, 40, -40, 40, 0., 10000.
  - hres, vres** [floats] horizontal and vertical grid resolution [m] Defaults 1000., 500.
  - latorig, lonorig, altorig** [floats] latitude and longitude of grid origin [deg] and altitude of grid origin [m MSL] Defaults the latitude, longitude and altitude of the radar
  - wfunc** [str. Dataset keyword] the weighting function used to combine the radar gates close to a grid point. Possible values BARNES, CRESSMAN, NEAREST\_NEIGHBOUR Default NEAREST\_NEIGHBOUR
  - roif\_func** [str. Dataset keyword] the function used to compute the region of interest. Possible values: dist\_beam, constant
  - roi** [float. Dataset keyword] the (minimum) radius of the region of interest in m. Default half the largest resolution

**beamwidth** [float. Dataset keyword] the radar antenna beamwidth [deg]. If None that of the key `radar_beam_width_h` in attribute `instrument_parameters` of the radar object will be used. If the key or the attribute are not present a default 1 deg value will be used

**beam\_spacing** [float. Dataset keyword] the beam spacing, i.e. the ray angle resolution [deg]. If None, that of the attribute `ray_angle_res` of the radar object will be used. If the attribute is None a default 1 deg value will be used

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the gridded data

**ind\_rad** [int] radar index

`pyrad.proc.process_grid.process_grid_point` (*procstatus, dscfg, radar\_list=None*)

Obtains the grid data at a point location.

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The data type where we want to extract the point measurement

**latlon** [boolean. Dataset keyword] if True position is obtained from latitude, longitude information, otherwise position is obtained from grid index (*iz, iy, ix*).

**lon** [float. Dataset keyword] the longitude [deg]. Use when `latlon` is True.

**lat** [float. Dataset keyword] the latitude [deg]. Use when `latlon` is True.

**alt** [float. Dataset keyword] altitude [m MSL]. Use when `latlon` is True.

**iz, iy, ix** [int. Dataset keyword] The grid indices. Use when `latlon` is False

**latlonTol** [float. Dataset keyword] latitude-longitude tolerance to determine which grid point to use [deg]

**altTol** [float. Dataset keyword] Altitude tolerance to determine which grid point to use [deg]

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the data and metadata at the point of interest

**ind\_rad** [int] radar index

`pyrad.proc.process_grid.process_grid_time_stats` (*procstatus, dscfg, radar\_list=None*)

computes the temporal statistics of a field

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**period** [float. Dataset keyword] the period to average [s]. If -1 the statistics are going to be performed over the entire data. Default 3600.

**start\_average** [float. Dataset keyword] when to start the average [s from midnight UTC]. Default 0.

**lin\_trans: int. Dataset keyword** If 1 apply linear transformation before averaging

**use\_nan** [bool. Dataset keyword] If true non valid data will be used

**nan\_value** [float. Dataset keyword] The value of the non valid data. Default 0

**stat: string. Dataset keyword** Statistic to compute: Can be mean, std, cov, min, max. Default mean

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_grid.process_grid_time_stats2(procstatus, dscfg, radar_list=None)`  
computes the temporal mean of a field

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**period** [float. Dataset keyword] the period to average [s]. If -1 the statistics are going to be performed over the entire data. Default 3600.

**start\_average** [float. Dataset keyword] when to start the average [s from midnight UTC]. Default 0.

**stat: string. Dataset keyword** Statistic to compute: Can be median, mode, percentileXX

**use\_nan** [bool. Dataset keyword] If true non valid data will be used

**nan\_value** [float. Dataset keyword] The value of the non valid data. Default 0

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_grid.process_raw_grid(procstatus, dscfg, radar_list=None)`  
Dummy function that returns the initial input data set

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index





## PYRAD.PROC.PROCESS\_INTERCOMP

Functions used in the inter-comparison between radars

<i>process_time_stats</i> (procstatus, dscfg[, ...])	computes the temporal statistics of a field
<i>process_time_stats2</i> (procstatus, dscfg[, ...])	computes the temporal mean of a field
<i>process_time_avg</i> (procstatus, dscfg[, radar_list])	computes the temporal mean of a field
<i>process_weighted_time_avg</i> (procstatus, dscfg)	computes the temporal mean of a field weighted by the reflectivity
<i>process_time_avg_flag</i> (procstatus, dscfg[, ...])	computes a flag field describing the conditions of the data used while averaging
<i>process_colocated_gates</i> (procstatus, dscfg[, ...])	Find colocated gates within two radars
<i>process_intercomp</i> (procstatus, dscfg[, ...])	intercomparison between two radars
<i>process_intercomp_time_avg</i> (procstatus, dscfg)	intercomparison between the average reflectivity of two radars
<i>process_fields_diff</i> (procstatus, dscfg[, ...])	Computes the field difference between RADAR001 and radar002, i.e.
<i>process_intercomp_fields</i> (procstatus, dscfg)	intercomparison between two radars

pyrad.proc.process\_intercomp.**process\_colocated\_gates** (*procstatus*, *dscfg*,  
*radar\_list=None*)  
Find colocated gates within two radars

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
  - datatype** [list of string. Dataset keyword] The input data types
  - h\_tol** [float. Dataset keyword] Tolerance in altitude difference between radar gates [m]. Default 100.
  - latlon\_tol** [float. Dataset keyword] Tolerance in latitude and longitude position between radar gates [deg]. Default 0.0005
  - vol\_d\_tol** [float. Dataset keyword] Tolerance in pulse volume diameter [m]. Default 100.
  - vismin** [float. Dataset keyword] Minimum visibility [percent]. Default None.
  - hmin** [float. Dataset keyword] Minimum altitude [m MSL]. Default None.
  - hmax** [float. Dataset keyword] Maximum altitude [m MSL]. Default None.
  - rmin** [float. Dataset keyword] Minimum range [m]. Default None.

**rmax** [float. Dataset keyword] Maximum range [m]. Default None.

**elmin** [float. Dataset keyword] Minimum elevation angle [deg]. Default None.

**elmax** [float. Dataset keyword] Maximum elevation angle [deg]. Default None.

**azrad1min** [float. Dataset keyword] Minimum azimuth angle [deg] for radar 1. Default None.

**azrad1max** [float. Dataset keyword] Maximum azimuth angle [deg] for radar 1. Default None.

**azrad2min** [float. Dataset keyword] Minimum azimuth angle [deg] for radar 2. Default None.

**azrad2max** [float. Dataset keyword] Maximum azimuth angle [deg] for radar 2. Default None.

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [radar object] radar object containing the flag field

**ind\_rad** [int] radar index

`pyrad.proc.process_intercomp.process_fields_diff(procstatus, dscfg, radar_list=None)`

Computes the field difference between RADAR001 and radar002, i.e. RADAR001-RADAR002. Assumes both radars have the same geometry

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing a radar object containing the field differences

**ind\_rad** [int] radar index

`pyrad.proc.process_intercomp.process_intercomp(procstatus, dscfg, radar_list=None)`

intercomparison between two radars

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**coloc\_data\_dir** [string. Dataset keyword] name of the directory containing the csv file with colocated data

**coloc\_radars\_name** [string. Dataset keyword] string identifying the radar names

**azi\_tol** [float. Dataset keyword] azimuth tolerance between the two radars. Default 0.5 deg

**ele\_tol** [float. Dataset keyword] elevation tolerance between the two radars. Default 0.5 deg

**rng\_tol** [float. Dataset keyword] range tolerance between the two radars. Default 50 m

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing a dictionary with intercomparison data and the key “final” which contains a boolean that is true when all volumes have been processed

**ind\_rad** [int] radar index

`pyrad.proc.process_intercomp.process_intercomp_fields` (*procstatus*, *dscfg*,  
*radar\_list=None*)

intercomparison between two radars

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing a dictionary with intercomparison data

**ind\_rad** [int] radar index

`pyrad.proc.process_intercomp.process_intercomp_time_avg` (*procstatus*, *dscfg*,  
*radar\_list=None*)

intercomparison between the average reflectivity of two radars

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**coloc\_data\_dir** [string. Dataset keyword] name of the directory containing the csv file with colocated data

**coloc\_radars\_name** [string. Dataset keyword] string identifying the radar names

**azi\_tol** [float. Dataset keyword] azimuth tolerance between the two radars. Default 0.5 deg

**ele\_tol** [float. Dataset keyword] elevation tolerance between the two radars. Default 0.5 deg

**rng\_tol** [float. Dataset keyword] range tolerance between the two radars. Default 50 m

**clt\_max** [int. Dataset keyword] maximum number of samples that can be clutter contaminated. Default 100 i.e. all

**phi\_excess\_max** [int. Dataset keyword] maximum number of samples that can have excess instantaneous PhiDP. Default 100 i.e. all

**non\_rain\_max** [int. Dataset keyword] maximum number of samples that can be no rain. Default 100 i.e. all

**phi\_avg\_max** [float. Dataset keyword] maximum average PhiDP allowed. Default 600 deg i.e. any

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing a dictionary with intercomparison data and the key “final” which contains a boolean that is true when all volumes have been processed

**ind\_rad** [int] radar index

`pyrad.proc.process_intercomp.process_time_avg(procstatus, dscfg, radar_list=None)`  
computes the temporal mean of a field

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**period** [float. Dataset keyword] the period to average [s]. Default 3600.

**start\_average** [float. Dataset keyword] when to start the average [s from midnight UTC]. Default 0.

**lin\_trans: int. Dataset keyword** If 1 apply linear transformation before averaging

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_intercomp.process_time_avg_flag(procstatus, dscfg, radar_list=None)`  
computes a flag field describing the conditions of the data used while averaging

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**period** [float. Dataset keyword] the period to average [s]. Default 3600.

**start\_average** [float. Dataset keyword] when to start the average [s from midnight UTC]. Default 0.

**phidpmax: float. Dataset keyword** maximum PhiDP

**beamwidth** [float. Dataset keyword] the antenna beamwidth [deg]. If None that of the keys `radar_beam_width_h` or `radar_beam_width_v` in attribute `instrument_parameters` of the radar object will be used. If the key or the attribute are not present the beamwidth will be set to None

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [Radar] radar object

**ind\_rad** [int] radar index

`pyrad.proc.process_intercomp.process_time_stats(procstatus, dscfg, radar_list=None)`  
computes the temporal statistics of a field

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**period** [float. Dataset keyword] the period to average [s]. If -1 the statistics are going to be performed over the entire data. Default 3600.

**start\_average** [float. Dataset keyword] when to start the average [s from midnight UTC]. Default 0.

**lin\_trans: int. Dataset keyword** If 1 apply linear transformation before averaging

**use\_nan** [bool. Dataset keyword] If true non valid data will be used

**nan\_value** [float. Dataset keyword] The value of the non valid data. Default 0

**stat: string. Dataset keyword** Statistic to compute: Can be mean, std, cov, min, max. Default mean

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_intercomp.process_time_stats2(procstatus, dscfg, radar_list=None)`  
computes the temporal mean of a field

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**period** [float. Dataset keyword] the period to average [s]. If -1 the statistics are going to be performed over the entire data. Default 3600.

**start\_average** [float. Dataset keyword] when to start the average [s from midnight UTC]. Default 0.

**stat: string. Dataset keyword** Statistic to compute: Can be median, mode, percentileXX

**use\_nan** [bool. Dataset keyword] If true non valid data will be used

**nan\_value** [float. Dataset keyword] The value of the non valid data. Default 0

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_intercomp.process_weighted_time_avg(procstatus, dscfg, radar_list=None)`  
computes the temporal mean of a field weighted by the reflectivity

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**period** [float. Dataset keyword] the period to average [s]. Default 3600.

**start\_average** [float. Dataset keyword] when to start the average [s from midnight UTC].  
Default 0.

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [Radar] radar object

**ind\_rad** [int] radar index

## PYRAD.PROC.PROCESS\_IQ

Functions to processes IQ data.

<i>process_raw_iq</i> (procstatus, dscfg[, radar_list])	Dummy function that returns the initial input data set
<i>process_pol_variables_iq</i> (procstatus, dscfg)	Computes the polarimetric variables from the IQ data
<i>process_reflectivity_iq</i> (procstatus, dscfg[, ...])	Computes reflectivity from the IQ data
<i>process_st1_iq</i> (procstatus, dscfg[, radar_list])	Computes the statistical test one lag fluctuation from the horizontal or vertical IQ data
<i>process_st2_iq</i> (procstatus, dscfg[, radar_list])	Computes the statistical test two lag fluctuation from the horizontal or vertical IQ data
<i>process_wbn_iq</i> (procstatus, dscfg[, radar_list])	Computes the wide band noise from the horizontal or vertical IQ data
<i>process_differential_reflectivity_iq</i> (...)	Computes differential reflectivity from the horizontal and vertical IQ data
<i>process_mean_phase_iq</i> (procstatus, dscfg[, ...])	Computes the mean phase from the horizontal or vertical IQ data
<i>process_differential_phase_iq</i> (procstatus, dscfg)	Computes the differential phase from the horizontal and vertical IQ data
<i>process_rhoHV_iq</i> (procstatus, dscfg[, radar_list])	Computes RhoHV from the horizontal and vertical IQ data
<i>process_Doppler_velocity_iq</i> (procstatus, dscfg)	Compute the Doppler velocity from the spectral reflectivity
<i>process_Doppler_width_iq</i> (procstatus, dscfg)	Compute the Doppler spectrum width from the spectral reflectivity
<i>process_fft</i> (procstatus, dscfg[, radar_list])	Compute the Doppler spectra form the IQ data with a Fourier transform

pyrad.proc.process\_iq.**process\_Doppler\_velocity\_iq**(*procstatus*, *dscfg*,  
*radar\_list=None*)

Compute the Doppler velocity from the spectral reflectivity

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:
  - datatype** [list of string. Dataset keyword] The input data types
  - direction** [str] The convention used in the Doppler mean field. Can be negative\_away or negative\_towards
- radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_Doppler_width_iq(procstatus, dscfg, radar_list=None)`

Compute the Doppler spectrum width from the spectral reflectivity

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**subtract\_noise** [Bool] If True noise will be subtracted from the signals

**lag** [int] Time lag used in the denominator of the computation

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_differential_phase_iq(procstatus, dscfg, radar_list=None)`

Computes the differential phase from the horizontal and vertical IQ data

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**phase\_offset** [float. Dataset keyword] The system differential phase offset to remove

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_differential_reflectivity_iq(procstatus, dscfg, radar_list=None)`

Computes differential reflectivity from the horizontal and vertical IQ data

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**subtract\_noise** [Bool] If True noise will be subtracted from the signal

**lag** [int] The time lag to use in the estimators

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**



**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_fft(procstatus, dscfg, radar_list=None)`

Compute the Doppler spectra from the IQ data with a Fourier transform

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**window** [list of str] Parameters of the window used to obtain the spectra. The parameters are the ones corresponding to function `scipy.signal.windows.get_window`. It can also be `['None']`.

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_mean_phase_iq(procstatus, dscfg, radar_list=None)`

Computes the mean phase from the horizontal or vertical IQ data

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_pol_variables_iq(procstatus, dscfg, radar_list=None)`

Computes the polarimetric variables from the IQ data

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**subtract\_noise** [Bool] If True noise will be subtracted from the signal

**lag** [int] The time lag to use in the estimators

**direction** [str] The convention used in the Doppler mean field. Can be `negative_away` or `negative_towards`

**variables** [list of str] list of variables to compute. Default dBZ

**phase\_offset** [float. Dataset keyword] The system differential phase offset to remove

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_raw_iq(procstatus, dscfg, radar_list=None)`  
Dummy function that returns the initial input data set

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration  
**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_reflectivity_iq(procstatus, dscfg, radar_list=None)`  
Computes reflectivity from the IQ data

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
    **subtract\_noise** [Bool] If True noise will be subtracted from the signal  
**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_rhohv_iq(procstatus, dscfg, radar_list=None)`  
Computes RhoHV from the horizontal and vertical IQ data

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
    **subtract\_noise** [Bool] If True noise will be subtracted from the signal  
    **lag** [int] Time lag used in the computation  
**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_st1_iq(procstatus, dscfg, radar_list=None)`  
Computes the statistical test one lag fluctuation from the horizontal or vertical IQ data

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_st2_iq(procstatus, dscfg, radar_list=None)`

Computes the statistical test two lag fluctuation from the horizontal or vertical IQ data

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_iq.process_wbn_iq(procstatus, dscfg, radar_list=None)`

Computes the wide band noise from the horizontal or vertical IQ data

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index



## PYRAD.PROC.PROCESS\_MONITORING

Functions for monitoring of the polarimetric variables

<i>process_selfconsistency_kdp_phidp</i> (..., ...)	Computes specific differential phase and differential phase in rain using the selfconsistency between Zdr, Zh and KDP
<i>process_selfconsistency_bias</i> (procstatus, dscfg)	Estimates the reflectivity bias by means of the selfconsistency algorithm by Gourley
<i>process_estimate_phidp0</i> (procstatus, dscfg[, ...])	estimates the system differential phase offset at each ray
<i>process_rhoHV_rain</i> (procstatus, dscfg[, ...])	Keeps only suitable data to evaluate the 80 percentile of RhoHV in rain
<i>process_zdr_precip</i> (procstatus, dscfg[, ...])	Keeps only suitable data to evaluate the differential reflectivity in moderate rain or precipitation (for vertical scans)
<i>process_zdr_snow</i> (procstatus, dscfg[, radar_list])	Keeps only suitable data to evaluate the differential reflectivity in snow
<i>process_monitoring</i> (procstatus, dscfg[, ...])	computes monitoring statistics

pyrad.proc.process\_monitoring.**process\_estimate\_phidp0** (*procstatus*, *dscfg*,  
*radar\_list=None*  
 estimates the system differential phase offset at each ray

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
  - datatype** [list of string. Dataset keyword] The input data types
  - rmin** [float. Dataset keyword] The minimum range where to look for valid data [m]
  - rmax** [float. Dataset keyword] The maximum range where to look for valid data [m]
  - rcell** [float. Dataset keyword] The length of a continuous cell to consider it valid precip [m]
  - Zmin** [float. Dataset keyword] The minimum reflectivity [dBZ]
  - Zmax** [float. Dataset keyword] The maximum reflectivity [dBZ]
- radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

- new\_dataset** [dict] dictionary containing the output
- ind\_rad** [int] radar index

`pyrad.proc.process_monitoring.process_monitoring(procstatus, dscfg, radar_list=None)`  
computes monitoring statistics

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**step** [float. Dataset keyword] The width of the histogram bin. Default is None. In that case the default step in function `get_histogram_bins` is used

**max\_rays** [int. Dataset keyword] The maximum number of rays per sweep used when computing the histogram. If set above 0 the number of rays per sweep will be checked and if above **max\_rays** the last rays of the sweep will be removed

**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [Radar] radar object containing histogram data

**ind\_rad** [int] radar index

`pyrad.proc.process_monitoring.process_rhohv_rain(procstatus, dscfg, radar_list=None)`  
Keeps only suitable data to evaluate the 80 percentile of RhoHV in rain

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**rmin** [float. Dataset keyword] minimum range where to look for rain [m]. Default 1000.

**rmax** [float. Dataset keyword] maximum range where to look for rain [m]. Default 50000.

**Zmin** [float. Dataset keyword] minimum reflectivity to consider the bin as precipitation [dBZ]. Default 20.

**Zmax** [float. Dataset keyword] maximum reflectivity to consider the bin as precipitation [dBZ] Default 40.

**ml\_thickness** [float. Dataset keyword] assumed thickness of the melting layer. Default 700.

**fzl** [float. Dataset keyword] The default freezing level height. It will be used if no temperature field name is specified or the temperature field is not in the radar object. Default 2000.

**radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_monitoring.process_selfconsistency_bias(procstatus, dscfg, radar_list=None)`  
Estimates the reflectivity bias by means of the selfconsistency algorithm by Gourley

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**parametrization** [str] The type of parametrization for the self-consistency curves. Can be 'None', 'Gourley', 'Wolfensberger', 'Louf', 'Gorgucci' or 'Vaccarono' 'None' will use tables from config files. Default 'None'.

**fzl** [float. Dataset keyword] Default freezing level height. Default 2000.

**rsmooth** [float. Dataset keyword] length of the smoothing window [m]. Default 2000.

**min\_rhohv** [float. Dataset keyword] minimum valid RhoHV. Default 0.92

**filter\_rain** [Bool. Dataset keyword] If True the hydrometeor classification is used to filter out gates that are not rain. Default True

**max\_phidp** [float. Dataset keyword] maximum valid PhiDP [deg]. Default 20.

**ml\_thickness** [float. Dataset keyword] Melting layer thickness [m]. Default 700.

**rcell** [float. Dataset keyword] length of continuous precipitation to consider the precipitation cell a valid phidp segment [m]. Default 15000.

**dphidp\_min** [float. Dataset keyword] minimum phase shift [deg]. Default 2.

**dphidp\_max** [float. Dataset keyword] maximum phase shift [deg]. Default 16.

**frequency** [float. Dataset keyword] the radar frequency [Hz]. If None that of the key frequency in attribute instrument\_parameters of the radar object will be used. If the key or the attribute are not present the selfconsistency will not be computed

**check\_wet\_radome** [Bool. Dataset keyword] if True the average reflectivity of the closest gates to the radar is going to be check to find out whether there is rain over the radome. If there is rain no bias will be computed. Default True.

**wet\_radome\_refl** [Float. Dataset keyword] Average reflectivity [dBZ] of the gates close to the radar to consider the radome as wet. Default 25.

**wet\_radome\_rng\_min, wet\_radome\_rng\_max** [Float. Dataset keyword] Min and max range [m] of the disk around the radar used to compute the average reflectivity to determine whether the radome is wet. Default 2000 and 4000.

**wet\_radome\_ngates\_min** [int] Minimum number of valid gates to consider that the radome is wet. Default 180

**valid\_gates\_only** [Bool] If True the reflectivity bias obtained for each valid ray is going to be assigned only to gates of the segment used. That will give more weight to longer segments when computing the total bias. Default False

**keep\_points** [Bool] If True the ZDR, ZH and KDP of the gates used in the self-consistency algorithm are going to be stored for further analysis. Default False

**rkdip** [float] The length of the window used to compute KDP with the single window least square method [m]. Default 6000.

**radar\_list** [list of Radar objects] Optional. list of radar objects

## Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_monitoring.process_selfconsistency_kdp_phidp` (*procstatus*,  
*dscfg*,  
*radar\_list=None*)

Computes specific differential phase and differential phase in rain using the selfconsistency between Zdr, Zh and KDP

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of strings. Dataset keyword] The input data types

**parametrization** [str] The type of parametrization for the self-consistency curves. Can be 'None', 'Gourley', 'Wolfensberger', 'Louf', 'Gorgucci' or 'Vaccarono' 'None' will use tables from config files. Default 'None'.

**rsmooth** [float. Dataset keyword] length of the smoothing window [m]. Default 2000.

**min\_rhohv** [float. Dataset keyword] minimum valid RhoHV. Default 0.92

**filter\_rain** [Bool. Dataset keyword] If True the hydrometeor classification is used to filter out gates that are not rain. Default True

**max\_phidp** [float. Dataset keyword] maximum valid PhiDP [deg]. Default 20.

**ml\_thickness** [float. Dataset keyword] assumed melting layer thickness [m]. Default 700.

**fz1** [float. Dataset keyword] The default freezing level height. It will be used if no temperature field name is specified or the temperature field is not in the radar object. Default 2000.

**frequency** [float. Dataset keyword] the radar frequency [Hz]. If None that of the key frequency in attribute `instrument_parameters` of the radar object will be used. If the key or the attribute are not present the selfconsistency will not be computed

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_monitoring.process_zdr_precip` (*procstatus*, *dscfg*, *radar\_list=None*)

Keeps only suitable data to evaluate the differential reflectivity in moderate rain or precipitation (for vertical scans)

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**ml\_filter** [boolean. Dataset keyword] indicates if a filter on data in and above the melting layer is applied. Default True.

**rmin** [float. Dataset keyword] minimum range where to look for rain [m]. Default 1000.

**rmax** [float. Dataset keyword] maximum range where to look for rain [m]. Default 50000.

**Zmin** [float. Dataset keyword] minimum reflectivity to consider the bin as precipitation [dBZ]. Default 20.



**Zmax** [float. Dataset keyword] maximum reflectivity to consider the bin as precipitation [dBZ] Default 22.

**RhoHVmin** [float. Dataset keyword] minimum RhoHV to consider the bin as precipitation Default 0.97

**PhiDPmax** [float. Dataset keyword] maximum PhiDP to consider the bin as precipitation [deg] Default 10.

**elmax** [float. Dataset keyword] maximum elevation angle where to look for precipitation [deg] Default None.

**ml\_thickness** [float. Dataset keyword] assumed thickness of the melting layer. Default 700.

**fzl** [float. Dataset keyword] The default freezing level height. It will be used if no temperature field name is specified or the temperature field is not in the radar object. Default 2000.

**radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_monitoring.process_zdr_snow(procstatus, dscfg, radar_list=None)`

Keeps only suitable data to evaluate the differential reflectivity in snow

### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**rmin** [float. Dataset keyword] minimum range where to look for rain [m]. Default 1000.

**rmax** [float. Dataset keyword] maximum range where to look for rain [m]. Default 50000.

**Zmin** [float. Dataset keyword] minimum reflectivity to consider the bin as snow [dBZ]. Default 0.

**Zmax** [float. Dataset keyword] maximum reflectivity to consider the bin as snow [dBZ] Default 30.

**SNRmin** [float. Dataset keyword] minimum SNR to consider the bin as snow [dB]. Default 10.

**SNRmax** [float. Dataset keyword] maximum SNR to consider the bin as snow [dB] Default 50.

**RhoHVmin** [float. Dataset keyword] minimum RhoHV to consider the bin as snow Default 0.97

**PhiDPmax** [float. Dataset keyword] maximum PhiDP to consider the bin as snow [deg] Default 10.

**elmax** [float. Dataset keyword] maximum elevation angle where to look for snow [deg] Default None.

**KDPmax** [float. Dataset keyword] maximum KDP to consider the bin as snow [deg] Default None

**TEMPmin** [float. Dataset keyword] minimum temperature to consider the bin as snow [deg C]. Default None

**TEMPmax** [float. Dataset keyword] maximum temperature to consider the bin as snow [deg C] Default None

**hydroclass** [list of ints. Dataset keyword] list of hydrometeor classes to keep for the analysis Default [2] (dry snow)

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

## PYRAD.PROC.PROCESS\_PHASE

Functions for PhiDP and KDP processing and attenuation correction

<i>process_correct_phidp0</i> (procstatus, dscfg[, ...])	corrects phidp of the system phase
<i>process_smooth_phidp_single_window</i> (...[, ...])	corrects phidp of the system phase and smoothes it using one window
<i>process_smooth_phidp_double_window</i> (...[, ...])	corrects phidp of the system phase and smoothes it using one window
<i>process_kdp_leastsquare_single_window</i> (...[, ...])	Computes specific differential phase using a piecewise least square method
<i>process_kdp_leastsquare_double_window</i> (...[, ...])	Computes specific differential phase using a piecewise least square method
<i>process_phidp_kdp_Vulpiani</i> (procstatus, dscfg)	Computes specific differential phase and differential phase using the method developed by Vulpiani et al.
<i>process_phidp_kdp_Kalman</i> (procstatus, dscfg)	Computes specific differential phase and differential phase using the Kalman filter as proposed by Schneebeli et al.
<i>process_phidp_kdp_Maesaka</i> (procstatus, dscfg)	Estimates PhiDP and KDP using the method by Maesaka.
<i>process_phidp_kdp_lp</i> (procstatus, dscfg[, ...])	Estimates PhiDP and KDP using a linear programming algorithm.
<i>process_selfconsistency_kdp_phidp</i>	
<i>process_selfconsistency_bias</i>	
<i>process_attenuation</i> (procstatus, dscfg[, ...])	Computes specific attenuation and specific differential attenuation using the Z-Phi method and corrects reflectivity and differential reflectivity

pyrad.proc.process\_phase.**process\_attenuation** (*procstatus, dscfg, radar\_list=None*)

Computes specific attenuation and specific differential attenuation using the Z-Phi method and corrects reflectivity and differential reflectivity

### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**ATT\_METHOD** [float. Dataset keyword] The attenuation estimation method used. One of the following: ZPhi, Philin

**fz1** [float. Dataset keyword] The default freezing level height. It will be used if no temper-

ature field name is specified or the temperature field is not in the radar object. Default 2000.

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_phase.process_correct_phidp0` (*procstatus, dscfg, radar\_list=None*)  
corrects phidp of the system phase

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**rmin** [float. Dataset keyword] The minimum range where to look for valid data [m]

**rmax** [float. Dataset keyword] The maximum range where to look for valid data [m]

**rcell** [float. Dataset keyword] The length of a continuous cell to consider it valid precip [m]

**Zmin** [float. Dataset keyword] The minimum reflectivity [dBZ]

**Zmax** [float. Dataset keyword] The maximum reflectivity [dBZ]

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_phase.process_kdp_leastsquare_double_window` (*procstatus, dscfg, radar\_list=None*)

Computes specific differential phase using a piecewise least square method

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**rwinds** [float. Dataset keyword] The length of the short segment for the least square method [m]

**rwindl** [float. Dataset keyword] The length of the long segment for the least square method [m]

**Zthr** [float. Dataset keyword] The threshold defining which estimated data to use [dBZ]

**vectorize** [Bool. Dataset keyword] Whether to vectorize the KDP processing. Default false

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_phase.process_kdp_leastsquare_single_window` (*procstatus*,  
*dscfg*,  
*radar\_list=None*)

Computes specific differential phase using a piecewise least square method

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
    **rwind** [float. Dataset keyword] The length of the segment for the least square method [m]  
    **vectorize** [bool. Dataset keyword] Whether to vectorize the KDP processing. Default false  
**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_phase.process_phidp_kdp_Kalman` (*procstatus*,  
*dscfg*,  
*radar\_list=None*)

Computes specific differential phase and differential phase using the Kalman filter as proposed by Schneebeli et al. The data is assumed to be clutter free and continous

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
    **parallel** [boolean. Dataset keyword] if set use parallel computing  
    **get\_phidp** [boolean. Dataset keyword] if set the PhiDP computed by integrating the resultant KDP is added to the radar field  
    **frequency** [float. Dataset keyword] the radar frequency [Hz]. If None that of the key frequency in attribute `instrument_parameters` of the radar object will be used. If the key or the attribute are not present it will be assumed that the radar is C band  
**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_phase.process_phidp_kdp_Maesaka` (*procstatus*,  
*dscfg*,  
*radar\_list=None*)

Estimates PhiDP and KDP using the method by Maesaka. This method only retrieves data in rain (i.e. below the melting layer)

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
    **datatype** [list of string. Dataset keyword] The input data types  
    **rmin** [float. Dataset keyword] The minimum range where to look for valid data [m]

**rmax** [float. Dataset keyword] The maximum range where to look for valid data [m]  
**rcell** [float. Dataset keyword] The length of a continuous cell to consider it valid precip [m]  
**Zmin** [float. Dataset keyword] The minimum reflectivity [dBZ]  
**Zmax** [float. Dataset keyword] The maximum reflectivity [dBZ]  
**fzl** [float. Dataset keyword] The freezing level height [m]. Default 2000.  
**ml\_thickness** [float. Dataset keyword] The melting layer thickness in meters. Default 700.  
**beamwidth** [float. Dataset keyword] the antenna beamwidth [deg]. If None that of the keys `radar_beam_width_h` or `radar_beam_width_v` in attribute `instrument_parameters` of the radar object will be used. If the key or the attribute are not present the beamwidth will be set to None

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_phase.process_phidp_kdp_Vulpiani` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Computes specific differential phase and differential phase using the method developed by Vulpiani et al. The data is assumed to be clutter free and monotonous

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**rwind** [float. Dataset keyword] The length of the segment [m]

**n\_iter** [int. Dataset keyword] number of iterations

**interp** [boolean. Dataset keyword] if set non valid values are interpolated using neighbouring valid values

**parallel** [boolean. Dataset keyword] if set use parallel computing

**get\_phidp** [boolean. Dataset keyword] if set the PhiDP computed by integrating the resultant KDP is added to the radar field

**frequency** [float. Dataset keyword] the radar frequency [Hz]. If None that of the key `frequency` in attribute `instrument_parameters` of the radar object will be used. If the key or the attribute are not present it will be assumed that the radar is C band

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_phase.process_phidp_kdp_lp` (*procstatus*, *dscfg*, *radar\_list=None*)

Estimates PhiDP and KDP using a linear programming algorithm. This method only retrieves data in rain (i.e. below the melting layer)

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**fzl** [float. Dataset keyword] The freezing level height [m]. Default 2000.

**ml\_thickness** [float. Dataset keyword] The melting layer thickness in meters. Default 700.

**beamwidth** [float. Dataset keyword] the antenna beamwidth [deg]. If None that of the keys radar\_beam\_width\_h or radar\_beam\_width\_v in attribute instrument\_parameters of the radar object will be used. If the key or the attribute are not present the beamwidth will be set to None

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_phase.process_smooth_phidp_double_window` (*procstatus*, *dscfg*,  
*radar\_list=None*)

corrects phidp of the system phase and smoothes it using one window

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**rmin** [float. Dataset keyword] The minimum range where to look for valid data [m]

**rmax** [float. Dataset keyword] The maximum range where to look for valid data [m]

**rcell** [float. Dataset keyword] The length of a continuous cell to consider it valid precip [m]

**rwinds** [float. Dataset keyword] The length of the short smoothing window [m]

**rwindl** [float. Dataset keyword] The length of the long smoothing window [m]

**Zmin** [float. Dataset keyword] The minimum reflectivity [dBZ]

**Zmax** [float. Dataset keyword] The maximum reflectivity [dBZ]

**Zthr** [float. Dataset keyword] The threshold defining wich smoothed data to used [dBZ]

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_phase.process_smooth_phidp_single_window` (*procstatus*, *dscfg*,  
*radar\_list=None*)

corrects phidp of the system phase and smoothes it using one window

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**rmin** [float. Dataset keyword] The minimum range where to look for valid data [m]  
**rmax** [float. Dataset keyword] The maximum range where to look for valid data [m]  
**rcell** [float. Dataset keyword] The length of a continuous cell to consider it valid precip [m]  
**rwind** [float. Dataset keyword] The length of the smoothing window [m]  
**Zmin** [float. Dataset keyword] The minimum reflectivity [dBZ]  
**Zmax** [float. Dataset keyword] The maximum reflectivity [dBZ]  
**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index



## PYRAD.PROC.PROCESS\_RETRIEVE

Functions for retrieving new moments and products

<i>process_ccor</i> (procstatus, dscfg[, radar_list])	Computes the Clutter Correction Ratio, i.e.
<i>process_signal_power</i> (procstatus, dscfg[, ...])	Computes the signal power in dBm
<i>process_rcs_pr</i> (procstatus, dscfg[, radar_list])	Computes the radar cross-section (assuming a point target) from radar reflectivity by first computing the received power and then the RCS from it.
<i>process_rcs</i> (procstatus, dscfg[, radar_list])	Computes the radar cross-section (assuming a point target) from radar reflectivity.
<i>process_vol_refl</i> (procstatus, dscfg[, radar_list])	Computes the volumetric reflectivity in $10\log_{10}(\text{cm}^2 \text{ km}^{-3})$
<i>process_snr</i> (procstatus, dscfg[, radar_list])	Computes SNR
<i>process_l</i> (procstatus, dscfg[, radar_list])	Computes L parameter
<i>process_cdr</i> (procstatus, dscfg[, radar_list])	Computes Circular Depolarization Ratio
<i>process_rainrate</i> (procstatus, dscfg[, radar_list])	Estimates rainfall rate from polarimetric moments
<i>process_rainfall_accumulation</i> (procstatus, dscfg)	Computes rainfall accumulation fields
<i>process_bird_density</i> (procstatus, dscfg[, ...])	Computes the bird density from the volumetric reflectivity

`pyrad.proc.process_retrieve.process_bird_density` (*procstatus, dscfg, radar\_list=None*)  
Computes the bird density from the volumetric reflectivity

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
  - datatype** [list of string. Dataset keyword] The input data types
  - sigma\_bird** [float. Dataset keyword] The bird radar cross section
- radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

- new\_dataset** [dict] dictionary containing the output
- ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_ccor` (*procstatus, dscfg, radar\_list=None*)  
Computes the Clutter Correction Ratio, i.e. the ratio between the signal without Doppler filtering and the signal with Doppler filtering

### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
     **datatype** [list of string. Dataset keyword] The input data types  
**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_cdr (procstatus, dscfg, radar_list=None)`  
 Computes Circular Depolarization Ratio

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
     **datatype** [string. Dataset keyword] The input data type  
**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_l (procstatus, dscfg, radar_list=None)`  
 Computes L parameter

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
     **datatype** [string. Dataset keyword] The input data type  
**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output  
**ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_rainfall_accumulation (procstatus, dscfg, radar_list=None)`  
 Computes rainfall accumulation fields

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing  
**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:  
     **datatype** [list of string. Dataset keyword] The input data types  
     **period** [float. Dataset keyword] the period to average [s]. If -1 the statistics are going to be performed over the entire data. Default 3600.  
     **start\_average** [float. Dataset keyword] when to start the average [s from midnight UTC]. Default 0.

**use\_nan** [bool. Dataset keyword] If true non valid data will be used

**nan\_value** [float. Dataset keyword] The value of the non valid data. Default 0

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_rainrate(procstatus, dscfg, radar_list=None)`

Estimates rainfall rate from polarimetric moments

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The input data type

**RR\_METHOD** [string. Dataset keyword] The rainfall rate estimation method. One of the following: Z, ZPoly, KDP, A, ZKDP, ZA, hydro

**alpha, beta** [float] factor and exponent of the R-Var power law  $R = \alpha * \text{Var}^{\text{Beta}}$ . Default value depending on RR\_METHOD. Z (0.0376, 0.6112), KDP (None, None), A (None, None)

**alphaz, betaz** [float] factor and exponent of the R-Z power law  $R = \alpha * Z^{\text{Beta}}$ . Default value (0.0376, 0.6112)

**alphazr, betazr** [float] factor and exponent of the R-Z power law  $R = \alpha * Z^{\text{Beta}}$  applied to rain in method hydro. Default value (0.0376, 0.6112)

**alphazs, betazs** [float] factor and exponent of the R-Z power law  $R = \alpha * Z^{\text{Beta}}$  applied to solid precipitation in method hydro. Default value (0.1, 0.5)

**alphakdp, betakdp** [float] factor and exponent of the R-KDP power law  $R = \alpha * \text{KDP}^{\text{Beta}}$ . Default value (None, None)

**alphaa, betaa** [float] factor and exponent of the R-Ah power law  $R = \alpha * \text{Ah}^{\text{Beta}}$ . Default value (None, None)

**thresh** [float] In hybrid methods, Rainfall rate threshold at which the retrieval method used changes [mm/h]. Default value depending on RR\_METHOD. ZKDP 10, ZA 10, hydro 10

**mp\_factor** [float] Factor by which the Z-R relation is multiplied in the melting layer in method hydro. Default 0.6

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_rcs(procstatus, dscfg, radar_list=None)`

Computes the radar cross-section (assuming a point target) from radar reflectivity.

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**kw2** [float. Dataset keyword] The water constant

**pulse\_width** [float. Dataset keyword] The pulse width [s]

**beamwidthv** [float. Global keyword] The vertical polarization antenna beamwidth [deg].  
Used if input is vertical reflectivity

**beamwidthh** [float. Global keyword] The horizontal polarization antenna beamwidth  
[deg]. Used if input is horizontal reflectivity

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_rcs_pr(procstatus, dscfg, radar_list=None)`

Computes the radar cross-section (assuming a point target) from radar reflectivity by first computing the received power and then the RCS from it.

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**AntennaGainH, AntennaGainV** [float. Dataset keyword] The horizontal (vertical) polarization antenna gain [dB]. If None it will be obtained from the attribute `instrument_parameters` of the radar object

**txpwrh, txpwrv** [float. Dataset keyword] The transmitted power of the horizontal (vertical) channel [dBm]. If None it will be obtained from the attribute `radar_calibration` of the radar object

**mflossh, mflossv** [float. Dataset keyword] The matching filter losses of the horizontal (vertical) channel [dB]. If None it will be obtained from the attribute `radar_calibration` of the radar object. Defaults to 0

**radconsth, radconstv** [float. Dataset keyword] The horizontal (vertical) channel radar constant. If None it will be obtained from the attribute `radar_calibration` of the radar object

**lrxh, lrxv** [float. Global keyword] The horizontal (vertical) receiver losses from the antenna feed to the reference point. [dB] positive value. Default 0

**ltxh, ltxv** [float. Global keyword] The horizontal (vertical) transmitter losses from the output of the high power amplifier to the antenna feed. [dB] positive value. Default 0

**lradomeh, lradomev** [float. Global keyword] The 1-way dry radome horizontal (vertical) channel losses. [dB] positive value. Default 0.

**attg** [float. Dataset keyword] The gas attenuation [dB/km]. If none it will be obtained from the attribute `radar_calibration` of the radar object or assigned according to the radar frequency. Defaults to 0.

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_signal_power` (*procstatus, dscfg, radar\_list=None*)

Computes the signal power in dBm

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**mflossh, mflossv** [float. Dataset keyword] The matching filter losses of the horizontal (vertical) channel [dB]. If None it will be obtained from the attribute `radar_calibration` of the radar object. Defaults to 0

**radconsth, radconstv** [float. Dataset keyword] The horizontal (vertical) channel radar constant. If None it will be obtained from the attribute `radar_calibration` of the radar object

**lrxh, lrxv** [float. Global keyword] The horizontal (vertical) receiver losses from the antenna feed to the reference point. [dB] positive value. Default 0

**lradomeh, lradomev** [float. Global keyword] The 1-way dry radome horizontal (vertical) channel losses. [dB] positive value. Default 0.

**attg** [float. Dataset keyword] The gas attenuation [dB/km]. If none it will be obtained from the attribute `radar_calibration` of the radar object or assigned according to the radar frequency. Defaults to 0.

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_snr` (*procstatus, dscfg, radar\_list=None*)

Computes SNR

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The input data type

**output\_type** [string. Dataset keyword] The output data type. Either SNRh or SNRv

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_retrieve.process_vol_refl` (*procstatus, dscfg, radar\_list=None*)

Computes the volumetric reflectivity in  $10\log_{10}(\text{cm}^2 \text{ km}^{-3})$

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**freq** [float. Dataset keyword] The radar frequency

**kw** [float. Dataset keyword] The water constant

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

## PYRAD.PROC.PROCESS\_SPECTRA

Functions to processes spectral data.

<i>process_raw_spectra</i> (procstatus, dscfg[, ...])	Dummy function that returns the initial input data set
<i>process_ifft</i> (procstatus, dscfg[, radar_list])	Compute the Doppler spectrum width from the spectral reflectivity
<i>process_spectra_point</i> (procstatus, dscfg[, ...])	Obtains the spectra or IQ data at a point location.
<i>process_filter_0Doppler</i> (procstatus, dscfg[, ...])	Function to filter the 0-Doppler line bin and neighbours of the Doppler spectra
<i>process_filter_srhov</i> (procstatus, dscfg[, ...])	Filter Doppler spectra as a function of spectral RhoHV
<i>process_filter_spectra_noise</i> (procstatus, dscfg)	Filter the noise of the Doppler spectra by clipping any data below the noise level plus a margin
<i>process_spectra_ang_avg</i> (procstatus, dscfg[, ...])	Function to average the spectra over the rays.
<i>process_spectral_power</i> (procstatus, dscfg[, ...])	Computes the spectral power
<i>process_spectral_noise</i> (procstatus, dscfg[, ...])	Computes the spectral noise
<i>process_spectral_phase</i> (procstatus, dscfg[, ...])	Computes the spectral phase
<i>process_spectral_reflectivity</i> (procstatus, dscfg)	Computes spectral reflectivity
<i>process_spectral_differential_reflectivity</i> (procstatus, dscfg)	Computes spectral differential reflectivity
<i>process_spectral_differential_phase</i> (..., ...)	Computes the spectral differential phase
<i>process_spectral_rhov</i> (procstatus, dscfg[, ...])	Computes the spectral RhoHV
<i>process_pol_variables</i> (procstatus, dscfg[, ...])	Computes the polarimetric variables from the complex spectra
<i>process_noise_power</i> (procstatus, dscfg[, ...])	Computes the noise power from the spectra
<i>process_reflectivity</i> (procstatus, dscfg[, ...])	Computes reflectivity from the spectral reflectivity
<i>process_differential_reflectivity</i> (..., ...)	Computes differential reflectivity from the horizontal and vertical spectral reflectivity
<i>process_differential_phase</i> (procstatus, dscfg)	Computes the differential phase from the spectral differential phase and the spectral reflectivity
<i>process_rhov</i> (procstatus, dscfg[, radar_list])	Computes RhoHV from the complex spectras
<i>process_Doppler_velocity</i> (procstatus, dscfg)	Compute the Doppler velocity from the spectral reflectivity

Continued on next page

Table 1 – continued from previous page

---

<code>process_Doppler_width(procstatus, ...)</code>	<code>dscfg[,</code>	Compute the Doppler spectrum width from the spectral reflectivity
---	----------------------	---

---

`pyrad.proc.process_spectra.process_Doppler_velocity` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Compute the Doppler velocity from the spectral reflectivity

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_Doppler_width` (*procstatus*, *dscfg*, *radar\_list=None*)

Compute the Doppler spectrum width from the spectral reflectivity

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_differential_phase` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Computes the differential phase from the spectral differential phase and the spectral reflectivity

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_differential_reflectivity` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Computes differential reflectivity from the horizontal and vertical spectral reflectivity

**Parameters**



## Returns

pyrad.proc.process\_spectra.**process\_filter\_0Doppler** (*procstatus*, *dscfg*,  
*radar\_list=None*)  
 Function to filter the 0-Doppler line bin and neighbours of the Doppler spectra

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

- datatype** [list of string. Dataset keyword] The input data types
- filter\_width** [float] The Doppler filter width. Default 0.
- filter\_units** [str] Can be ‘m/s’ or ‘Hz’. Default ‘m/s’

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_filter_spectra_noise` (*procstatus*, *dscfg*, *radar\_list=None*)  
Filter the noise of the Doppler spectra by clipping any data below the noise level plus a margin

**proccstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

- datatype** [list of string. Dataset keyword] The input data types
- clipping\_level** [float] The clipping level [dB above noise level]. Default 10.

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_filter_srhoHV` (*procstatus, dscfg, radar\_list=None*)  
Filter Doppler spectra as a function of spectral RhoHV

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string, Dataset keyword] The input data types

**sRhoHV\_threshold** [float] Data with sRhoHV module above this threshold will be filtered.  
Default 1.

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_ifft` (*procstatus*, *dscfg*, *radar\_list=None*)

Compute the Doppler spectrum width from the spectral reflectivity

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_noise_power` (*procstatus*, *dscfg*, *radar\_list=None*)

Computes the noise power from the spectra

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**units** [str] The units of the returned signal. Can be 'ADU', 'dBADU' or 'dBm'

**navg** [int] Number of spectra averaged

**rmin** [int] Range from which the data is used to estimate the noise

**nnoise\_min** [int] Minimum number of samples to consider the estimated noise power valid

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_pol_variables` (*procstatus*, *dscfg*, *radar\_list=None*)

Computes the polarimetric variables from the complex spectra

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**subtract\_noise** [Bool] If True noise will be subtracted from the signal

**smooth\_window** [int or None] Size of the moving Gaussian smoothing window. If none no smoothing will be applied

**variables** [list of str] list of variables to compute. Default dBZ

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_raw_spectra(procstatus, dscfg, radar_list=None)`

Dummy function that returns the initial input data set

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_reflectivity(procstatus, dscfg, radar_list=None)`

Computes reflectivity from the spectral reflectivity

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_rhoHV(procstatus, dscfg, radar_list=None)`

Computes RhoHV from the complex spectras

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**subtract\_noise** [Bool] If True noise will be subtracted from the signal

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_spectra_ang_avg` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Function to average the spectra over the rays. This function is intended mainly for vertically pointing scans. The function assumes the volume is composed of a single sweep, it averages over the number of rays specified by the user and produces a single ray output.

#### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:
  - datatype** [list of string. Dataset keyword] The input data types
  - navg** [int] Number of spectra to average. If -1 all spectra will be averaged. Default -1.
  - radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

- new\_dataset** [dict] dictionary containing the output
- ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_spectra_point` (*procstatus*, *dscfg*, *radar\_list=None*)

Obtains the spectra or IQ data at a point location.

#### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
  - datatype** [string. Dataset keyword] The data type where we want to extract the point measurement
  - latlon** [boolean. Dataset keyword] if True position is obtained from latitude, longitude information, otherwise position is obtained from antenna coordinates (range, azimuth, elevation). Default False
  - truealt** [boolean. Dataset keyword] if True the user input altitude is used to determine the point of interest. if False use the altitude at a given radar elevation *ele* over the point of interest. Default True
  - lon** [float. Dataset keyword] the longitude [deg]. Use when *latlon* is True.
  - lat** [float. Dataset keyword] the latitude [deg]. Use when *latlon* is True.
  - alt** [float. Dataset keyword] altitude [m MSL]. Use when *latlon* is True. Default 0.
  - ele** [float. Dataset keyword] radar elevation [deg]. Use when *latlon* is False or when *latlon* is True and *truealt* is False
  - azi** [float. Dataset keyword] radar azimuth [deg]. Use when *latlon* is False
  - rng** [float. Dataset keyword] range from radar [m]. Use when *latlon* is False
  - AziTol** [float. Dataset keyword] azimuthal tolerance to determine which radar azimuth to use [deg]. Default 0.5
  - EleTol** [float. Dataset keyword] elevation tolerance to determine which radar elevation to use [deg]. Default 0.5
  - RngTol** [float. Dataset keyword] range tolerance to determine which radar bin to use [m]. Default 50.
- radar\_list** [list of Radar objects] Optional. list of radar objects

**Returns**

**new\_dataset** [dict] dictionary containing the data and metadata at the point of interest

**ind\_rad** [int] radar index

```
pyrad.proc.process_spectra.process_spectral_differential_phase(procstatus,  
                                                             dscfg,  
                                                             radar_list=None)
```

Computes the spectral differential phase

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

```
pyrad.proc.process_spectra.process_spectral_differential_reflectivity(procstatus,  
                                                                    dscfg,  
                                                                    radar_list=None)
```

Computes spectral differential reflectivity

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**subtract\_noise** [Bool] If True noise will be subtracted from the signal

**smooth\_window** [int or None] Size of the moving Gaussian smoothing window. If none no smoothing will be applied

**radar\_list** [list of spectra objects] Optional. list of spectra objects

**Returns**

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

```
pyrad.proc.process_spectra.process_spectral_noise(procstatus,  
                                                  dscfg,  
                                                  radar_list=None)
```

Computes the spectral noise

**Parameters**

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**units** [str] The units of the returned signal. Can be 'ADU', 'dBADU' or 'dBm'

**navg** [int] Number of spectra averaged

**rmin** [int] Range from which the data is used to estimate the noise

**nnoise\_min** [int] Minimum number of samples to consider the estimated noise power valid

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

pyrad.proc.process\_spectra.**process\_spectral\_phase** (*procstatus*, *dscfg*,  
*radar\_list=None*)

Computes the spectral phase

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

pyrad.proc.process\_spectra.**process\_spectral\_power** (*procstatus*, *dscfg*,  
*radar\_list=None*)

Computes the spectral power

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**units** [str] The units of the returned signal. Can be 'ADU', 'dBADU' or 'dBm'

**subtract\_noise** [Bool] If True noise will be subtracted from the signal

**smooth\_window** [int or None] Size of the moving Gaussian smoothing window. If none  
no smoothing will be applied

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

pyrad.proc.process\_spectra.**process\_spectral\_reflectivity** (*procstatus*, *dscfg*,  
*radar\_list=None*)

Computes spectral reflectivity

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**subtract\_noise** [Bool] If True noise will be subtracted from the signal

**smooth\_window** [int or None] Size of the moving Gaussian smoothing window. If none no smoothing will be applied

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index

`pyrad.proc.process_spectra.process_spectral_rhoHV` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Computes the spectral RhoHV

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted configuration keywords:

**datatype** [list of string. Dataset keyword] The input data types

**subtract\_noise** [Bool] If True noise will be subtracted from the signal

**radar\_list** [list of spectra objects] Optional. list of spectra objects

#### Returns

**new\_dataset** [dict] dictionary containing the output

**ind\_rad** [int] radar index





## PYRAD.PROC.PROCESS\_TIMESERIES

Functions to obtain time series of radar data

<i>process_point_measurement</i> (procstatus, dscfg)	Obtains the radar data at a point location.
<i>process_qvp</i> (procstatus, dscfg[, radar_list])	Computes quasi vertical profiles, by averaging over height levels PPI data.
<i>process_rqvp</i> (procstatus, dscfg[, radar_list])	Computes range defined quasi vertical profiles, by averaging over height levels PPI data.
<i>process_evp</i> (procstatus, dscfg[, radar_list])	Computes enhanced vertical profiles, by averaging over height levels PPI data.
<i>process_svp</i> (procstatus, dscfg[, radar_list])	Computes slanted vertical profiles, by averaging over height levels PPI data.
<i>process_time_height</i> (procstatus, dscfg[, ...])	Produces time height radar objects at a point of interest defined by latitude and longitude.

`pyrad.proc.process_timeseries.process_evp` (*procstatus*, *dscfg*, *radar\_list=None*)  
Computes enhanced vertical profiles, by averaging over height levels PPI data.

### Parameters

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
- datatype** [string. Dataset keyword] The data type where we want to extract the point measurement
  - lat, lon** [float] latitude and longitude of the point of interest [deg]
  - latlon\_tol** [float] tolerance in latitude and longitude in deg. Default 0.0005
  - delta\_rng, delta\_azi** [float] maximum range distance [m] and azimuth distance [degree] from the central point of the evp containing data to average. Default 5000. and 10.
  - hmax** [float] The maximum height to plot [m]. Default 10000.
  - hres** [float] The height resolution [m]. Default 250.
  - avg\_type** [str] The type of averaging to perform. Can be either “mean” or “median” Default “mean”
  - nvalid\_min** [int] Minimum number of valid points to consider the data valid when performing the averaging. Default 1
  - interp\_kind** [str] type of interpolation when projecting to vertical grid: ‘none’, or ‘nearest’, etc. Default ‘none’. ‘none’ will select from all data points within the regular grid height

bin the closest to the center of the bin. 'nearest' will select the closest data point to the center of the height bin regardless if it is within the height bin or not. Data points can be masked values If another type of interpolation is selected masked values will be eliminated from the data points before the interpolation

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the EVP and a keyboard stating whether the processing has finished or not.

**ind\_rad** [int] radar index

`pyrad.proc.process_timeseries.process_point_measurement` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Obtains the radar data at a point location.

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The data type where we want to extract the point measurement

**latlon** [boolean. Dataset keyword] if True position is obtained from latitude, longitude information, otherwise position is obtained from antenna coordinates (range, azimuth, elevation).

**truealt** [boolean. Dataset keyword] if True the user input altitude is used to determine the point of interest. if False use the altitude at a given radar elevation ele over the point of interest.

**lon** [float. Dataset keyword] the longitude [deg]. Use when latlon is True.

**lat** [float. Dataset keyword] the latitude [deg]. Use when latlon is True.

**alt** [float. Dataset keyword] altitude [m MSL]. Use when latlon is True.

**ele** [float. Dataset keyword] radar elevation [deg]. Use when latlon is False or when latlon is True and truealt is False

**azi** [float. Dataset keyword] radar azimuth [deg]. Use when latlon is False

**rng** [float. Dataset keyword] range from radar [m]. Use when latlon is False

**AziTol** [float. Dataset keyword] azimuthal tolerance to determine which radar azimuth to use [deg]

**EleTol** [float. Dataset keyword] elevation tolerance to determine which radar elevation to use [deg]

**RngTol** [float. Dataset keyword] range tolerance to determine which radar bin to use [m]

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the data and metadata at the point of interest

**ind\_rad** [int] radar index

`pyrad.proc.process_timeseries.process_qvp` (*procstatus*, *dscfg*, *radar\_list=None*)

Computes quasi vertical profiles, by averaging over height levels PPI data.

### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The data type where we want to extract the point measurement

**angle** [int or float] If the radar object contains a PPI volume, the sweep number to use, if it contains an RHI volume the elevation angle. Default 0.

**ang\_tol** [float] If the radar object contains an RHI volume, the tolerance in the elevation angle for the conversion into PPI

**hmax** [float] The maximum height to plot [m]. Default 10000.

**hres** [float] The height resolution [m]. Default 50

**avg\_type** [str] The type of averaging to perform. Can be either “mean” or “median” Default “mean”

**nvalid\_min** [int] Minimum number of valid points to accept average. Default 30.

**interp\_kind** [str] type of interpolation when projecting to vertical grid: ‘none’, or ‘nearest’, etc. Default ‘none’ ‘none’ will select from all data points within the regular grid height bin the closest to the center of the bin. ‘nearest’ will select the closest data point to the center of the height bin regardless if it is within the height bin or not. Data points can be masked values If another type of interpolation is selected masked values will be eliminated from the data points before the interpolation

**radar\_list** [list of Radar objects] Optional. list of radar objects

### Returns

**new\_dataset** [dict] dictionary containing the QVP and a keyboard stating whether the processing has finished or not.

**ind\_rad** [int] radar index

`pyrad.proc.process_timeseries.process_rqvp(procstatus, dscfg, radar_list=None)`

Computes range defined quasi vertical profiles, by averaging over height levels PPI data.

### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The data type where we want to extract the point measurement

**anglenr** [int] The sweep number to use. It assumes the radar volume consists on PPI scans

**hmax** [float] The maximum height to plot [m]. Default 10000.

**hres** [float] The height resolution [m]. Default 2.

**avg\_type** [str] The type of averaging to perform. Can be either “mean” or “median” Default “mean”

**nvalid\_min** [int] Minimum number of valid points to accept average. Default 30.

**interp\_kind** [str] type of interpolation when projecting to vertical grid: ‘none’, or ‘nearest’, etc. Default ‘nearest’ ‘none’ will select from all data points within the regular grid height bin the closest to the center of the bin. ‘nearest’ will select the closest data point to

the center of the height bin regardless if it is within the height bin or not. Data points can be masked values If another type of interpolation is selected masked values will be eliminated from the data points before the interpolation

**rmax** [float] ground range up to which the data is intended for use [m]. Default 50000.

**weight\_power** [float] Power  $p$  of the weighting function  $1/abs(grng-(rmax-1))^{**p}$  given to the data outside the desired range. -1 will set the weight to 0. Default 2.

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the QVP and a keyboard stating whether the processing has finished or not.

**ind\_rad** [int] radar index

`pyrad.proc.process_timeseries.process_svp(procstatus, dscfg, radar_list=None)`

Computes slanted vertical profiles, by averaging over height levels PPI data.

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The data type where we want to extract the point measurement

**angle** [int or float] If the radar object contains a PPI volume, the sweep number to use, if it contains an RHI volume the elevation angle. Default 0.

**ang\_tol** [float] If the radar object contains an RHI volume, the tolerance in the elevation angle for the conversion into PPI. Default 1.

**lat, lon** [float] latitude and longitude of the point of interest [deg]

**latlon\_tol** [float] tolerance in latitude and longitude in deg. Default 0.0005

**delta\_rng, delta\_azi** [float] maximum range distance [m] and azimuth distance [degree] from the central point of the svp containing data to average. Default 5000. and 10.

**hmax** [float] The maximum height to plot [m]. Default 10000.

**hres** [float] The height resolution [m]. Default 250.

**avg\_type** [str] The type of averaging to perform. Can be either “mean” or “median” Default “mean”

**nvalid\_min** [int] Minimum number of valid points to consider the data valid when performing the averaging. Default 1

**interp\_kind** [str] type of interpolation when projecting to vertical grid: ‘none’, or ‘nearest’, etc. Default ‘none’ ‘none’ will select from all data points within the regular grid height bin the closest to the center of the bin. ‘nearest’ will select the closest data point to the center of the height bin regardless if it is within the height bin or not. Data points can be masked values If another type of interpolation is selected masked values will be eliminated from the data points before the interpolation

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the svp and a keyboard stating whether the processing has finished or not.

**ind\_rad** [int] radar index

`pyrad.proc.process_timeseries.process_time_height` (*procstatus*, *dscfg*,  
*radar\_list=None*)

Produces time height radar objects at a point of interest defined by latitude and longitude. A time-height contains the evolution of the vertical structure of radar measurements above the location of interest.

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [string. Dataset keyword] The data type where we want to extract the point measurement

**lat, lon** [float] latitude and longitude of the point of interest [deg]

**latlon\_tol** [float] tolerance in latitude and longitude in deg. Default 0.0005

**hmax** [float] The maximum height to plot [m]. Default 10000.

**hres** [float] The height resolution [m]. Default 50

**interp\_kind** [str] type of interpolation when projecting to vertical grid: 'none', or 'nearest', etc. Default 'none' 'none' will select from all data points within the regular grid height bin the closest to the center of the bin. 'nearest' will select the closest data point to the center of the height bin regardless if it is within the height bin or not. Data points can be masked values If another type of interpolation is selected masked values will be eliminated from the data points before the interpolation

**radar\_list** [list of Radar objects] Optional. list of radar objects

#### Returns

**new\_dataset** [dict] dictionary containing the QVP and a keyboard stating whether the processing has finished or not.

**ind\_rad** [int] radar index



## PYRAD.PROC.PROCESS\_TRAJ

Trajectory functions. Functions to pass trajectory dataset data to the product generation functions.

<code>process_trajectory(procstatus, dscfg[, ...])</code>	Return trajectory
<code>process_traj_trt(procstatus, dscfg[, ...])</code>	Processes data according to TRT trajectory
<code>process_traj_trt_contour(procstatus, dscfg)</code>	Gets the TRT cell contour corresponding to each radar volume
<code>process_traj_lightning(procstatus, dscfg[, ...])</code>	Return time series according to lightning trajectory
<code>process_traj_atplane(procstatus, dscfg[, ...])</code>	Return time series according to trajectory
<code>process_traj_antenna_pattern(procstatus, dscfg)</code>	Process a new array of data volumes considering a plane trajectory.
<code>_get_ts_values_antenna_pattern(radar, ...)</code>	Get the time series values of a trajectory using a synthetic antenna pattern
<code>_get_contour_trt(radar, trajectory, voltime)</code>	Get the TRT cell contour corresponding to the current radar
<code>_get_gates(radar, az, el, rr, tt, trajdict)</code>	Find the gates of the radar object that have to be used to compute the data of a trajectory
<code>_get_gates_trt(radar, trajectory, voltime[, ...])</code>	Find the gates of the radar object that belong to a TRT cell
<code>_get_gates_antenna_pattern(radar_sel, ...[, ...])</code>	Find the gates of the radar object that have to be used to compute the data of a trajectory as seen by another radar system
<code>_get_closest_bin(az, el, rr, tt, radar, tdict)</code>	Get the radar bin closest to a certain trajectory position
<code>_sample_out_of_sector(az, el, rr, radar_sel, ...)</code>	Check if trajectory sample is within radar sector
<code>TargetRadar(latitude, longitude, altitude)</code>	A class for dummy target radar object

**class** `pyrad.proc.process_traj.TargetRadar` (*latitude, longitude, altitude*)

Bases: `object`

A class for dummy target radar object

### Attributes

**latitude, longitude, altitude** [float] Position of the dummy radar

**\_\_class\_\_**

alias of `builtins.type`

**\_\_delattr\_\_** (*self, name, /*)

Implement `delattr(self, name)`.

**\_\_dict\_\_** = `mappingproxy({'__module__': 'pyrad.proc.process_traj', '__doc__': '\n A c`

**`__dir__`** (*self*, /)  
Default dir() implementation.

**`__eq__`** (*self*, *value*, /)  
Return self==value.

**`__format__`** (*self*, *format\_spec*, /)  
Default object formatter.

**`__ge__`** (*self*, *value*, /)  
Return self>=value.

**`__getattr__`** (*self*, *name*, /)  
Return getattr(self, name).

**`__gt__`** (*self*, *value*, /)  
Return self>value.

**`__hash__`** (*self*, /)  
Return hash(self).

**`__init__`** (*self*, *latitude*, *longitude*, *altitude*)  
Initialize self. See help(type(self)) for accurate signature.

**`__init_subclass__`** ()  
This method is called when a class is subclassed.  
  
The default implementation does nothing. It may be overridden to extend subclasses.

**`__le__`** (*self*, *value*, /)  
Return self<=value.

**`__lt__`** (*self*, *value*, /)  
Return self<value.

**`__module__`** = 'pyrad.proc.process\_traj'

**`__ne__`** (*self*, *value*, /)  
Return self!=value.

**`__new__`** (\*args, \*\*kwargs)  
Create and return a new object. See help(type) for accurate signature.

**`__reduce__`** (*self*, /)  
Helper for pickle.

**`__reduce_ex__`** (*self*, *protocol*, /)  
Helper for pickle.

**`__repr__`** (*self*, /)  
Return repr(self).

**`__setattr__`** (*self*, *name*, *value*, /)  
Implement setattr(self, name, value).

**`__sizeof__`** (*self*, /)  
Size of object in memory, in bytes.

**`__str__`** (*self*, /)  
Return str(self).

**`__subclasshook__`** ()  
Abstract classes can override this to customize issubclass().



This is invoked early on by `abc.ABCMeta.__subclasscheck__()`. It should return `True`, `False` or `NotImplemented`. If it returns `NotImplemented`, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

#### **`__weakref__`**

list of weak references to the object (if defined)

`pyrad.proc.process_traj._get_closest_bin` (*az, el, rr, tt, radar, tdict*)

Get the radar bin closest to a certain trajectory position

#### **Parameters**

**az, el, rr** [floats] The trajectory position respect to the radar

**tt** [float] the trajectory time respect to the beginning of the radar scan

**radar** [radar object] the current radar object

**tdict** [dict] Dictionary containing trajectory parameters

#### **Returns**

**radar\_sel** [radar object] The selected radar (Current or one of the two previous ones)

**ray\_sel, rr\_ind** [int] The selected ray and range indices of the radar field

**el\_vec\_rnd, az\_vec\_rnd** [array of floats] The elevation and azimuth fields of the selected radar rounded to the first decimal

`pyrad.proc.process_traj._get_contour_trt` (*radar, trajectory, voltime, time\_tol=100.0*)

Get the TRT cell contour corresponding to the current radar

#### **Parameters**

**radar** [radar object] The radar containing

**trajectory** [trajectory object] Object containing the TRT cell position and dimensions

**voltime** [datetime object] The radar volume reference time

**time\_tol** [float] Time tolerance where to look for data [s]

#### **Returns**

**lon\_roi, lat\_roi** [array of floats] The lat/lon defining the TRT cell contour

`pyrad.proc.process_traj._get_gates` (*radar, az, el, rr, tt, trajdict, ang\_tol=1.2*)

Find the gates of the radar object that have to be used to compute the data of a trajectory

#### **Parameters**

**radar** [radar object] The radar containing

**az, el, rr** [floats] The trajectory position respect to the radar

**tt** [float] the trajectory time respect to the beginning of the radar scan

**trajdict** [dict] Dictionary containing the trajectory parameters

**ang\_tol** [float] Factor that multiplies the angle resolution. Used when determining the neighbouring rays

#### **Returns**

**radar\_sel** [radar object] The radar volume selected as closest to trajectory point

**ray\_sel, rr\_ind** [ints] ray and range indices of the radar gate closest to the trajectory position

**cell\_ind** [array of ints] indices of the surrounding rays

**rr\_min** [int] index of the minimum range of the surrounding gates

**rr\_max** [int] index of the maximum range of the surrounding gates

```
pyrad.proc.process_traj._get_gates_antenna_pattern(radar_sel, target_radar,
                                                    az, rr, tt, scan_angles,
                                                    alt_tol=1000.0, latlon_tol=0.04,
                                                    max_altitude=12000.0)
```

Find the gates of the radar object that have to be used to compute the data of a trajectory as seen by another radar system

#### Parameters

**radar\_sel** [radar object] The radar containing real data

**target\_radar** [radar object] The virtual radar

**az, rr** [floats] The trajectory position respect to the radar

**tt** [float] the trajectory time respect to the beginning of the radar scan

**scan\_angles** [array] The scan angles of the virtual radar object

**alt\_tol** [float] The tolerance in altitude [m]

**latlon\_tol** [float] The tolerance in latitude and longitude [deg]

**max\_altitude** [float] The maximum altitude where to look for radar data

#### Returns

**ray\_ind, rng\_ind** [array of ints] the indices of the radar data to use

**w\_ind** [array of ints] The indices of the one-dimensional antenna pattern to use

```
pyrad.proc.process_traj._get_gates_trt(radar, trajectory, voltime, time_tol=100.0,
                                       alt_min=None, alt_max=None, cell_center=False,
                                       latlon_tol=0.0005)
```

Find the gates of the radar object that belong to a TRT cell

#### Parameters

**radar** [radar object] The radar containing

**trajectory** [trajectory object] Object containing the TRT cell position and dimensions

**voltime** [datetime object] The radar volume reference time

**time\_tol** [float] Time tolerance where to look for data [s]

**alt\_min, alt\_max** [float] Minimum and maximum altitude where to look for data [m]

#### Returns

**inds\_ray, inds\_rng** [array of ints] The indices of the radar data inside the TRT cell

```
pyrad.proc.process_traj._get_ts_values_antenna_pattern(radar, trajectory, tdict,
                                                       traj_ind, field_names)
```

Get the time series values of a trajectory using a synthetic antenna pattern

#### Parameters

**radar** [radar object] The radar volume with the data

**trajectory** [trajectory object] The plane trajectory

**tdict** [dict] A dictionary containing parameters useful for trajectory computation

**traj\_ind** [array] The indices of trajectory data within the current radar volume time

**field\_names** [list of str] list of names of the radar field

#### Returns

**result** [Bool] A flag signaling whether radar data matching the trajectory was found

`pyrad.proc.process_traj._sample_out_of_sector` (*az, el, rr, radar\_sel, ray\_sel, rr\_ind, el\_vec\_rnd, az\_vec\_rnd*)

Check if trajectory sample is within radar sector

#### Parameters

**az, el, rr** [floats] The trajectory position respect to the radar

**radar\_sel** [radar object] The selected radar (Current or one of the two previous ones)

**ray\_sel, rr\_ind** [int] The selected ray and range indices of the radar field

**el\_vec\_rnd, az\_vec\_rnd** [array of floats] The elevation and azimuth fields of the selected radar rounded to the first decimal

#### Returns

**result** [bool] False if the sample is out of sector. True otherwise

`pyrad.proc.process_traj.process_traj_antenna_pattern` (*procstatus, dscfg, radar\_list=None, trajectory=None*)

Process a new array of data volumes considering a plane trajectory. As result a timeseries with the values transposed for a given antenna pattern is created. The result is created when the LAST flag is set.

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries]

**datatype** [list of string. Dataset keyword] The input data types

**antennaType** [str. Dataset keyword] Type of antenna of the radar we want to get the view from. Can be AZIMUTH, ELEVATION, LOWBEAM, HIGHBEAM

**par\_azimuth\_antenna** [dict. Global ekyword] Dictionary containing the parameters of the PAR azimuth antenna, i.e. name of the file with the antenna elevation pattern and fixed antenna angle

**par\_elevation\_antenna** [dict. Global keyword] Dictionary containing the parameters of the PAR elevation antenna, i.e. name of the file with the antenna azimuth pattern and fixed antenna angle

**asr\_lowbeam\_antenna** [dict. Global keyword] Dictionary containing the parameters of the ASR low beam antenna, i.e. name of the file with the antenna elevation pattern and fixed antenna angle

**asr\_highbeam\_antenna** [dict. Global keyword] Dictionary containing the parameters of the ASR high beam antenna, i.e. name of the file with the antenna elevation pattern and fixed antenna angle

**target\_radar\_pos** [dict. Global keyword] Dictionary containing the latitude, longitude and altitude of the radar we want to get the view from. If not specifying it will assume the radar is collocated

**range\_all** [Bool. Dataset keyword] If the real radar and the synthetic radar are co-located and this parameter is true the statistics are going to be computed using all the data from range 0 to the position of the plane. Default False

**rhi\_resolution** [Bool. Dataset keyword] Resolution of the synthetic RHI used to compute the data as viewed from the synthetic radar [deg]. Default 0.5

**max\_altitude** [float. Dataset keyword] Max altitude of the data to use when computing the view from the synthetic radar [m MSL]. Default 12000.

**latlon\_tol** [float. Dataset keyword] The tolerance in latitude and longitude to determine which synthetic radar gates are co-located with real radar gates [deg]. Default 0.04

**alt\_tol** [float. Dataset keyword] The tolerance in altitude to determine which synthetic radar gates are co-located with real radar gates [m]. Default 1000.

**pattern\_thres** [float. Dataset keyword] The minimum of the sum of the weights given to each value in order to consider the weighted quantile valid. It is related to the number of valid data points

**data\_is\_log** [dict. Dataset keyword] Dictionary specifying for each field if it is in log (True) or linear units (False). Default False

**use\_nans** [dict. Dataset keyword] Dictionary specifying whether the nans have to be used in the computation of the statistics for each field. Default False

**nan\_value** [dict. Dataset keyword] Dictionary with the value to use to substitute the NaN values when computing the statistics of each field. Default 0

**radar\_list** [list of Radar objects] Optional. list of radar objects

**trajectory** [Trajectory object] containing trajectory samples

#### Returns

**trajectory** [Trajectory object] Object holding time series

**ind\_rad** [int] radar index

`pyrad.proc.process_traj.process_traj_atplane` (*procstatus, dscfg, radar\_list=None, trajectory=None*)

Return time series according to trajectory

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**data\_is\_log** [dict. Dataset keyword] Dictionary specifying for each field if it is in log (True) or linear units (False). Default False

**ang\_tol** [float. Dataset keyword] Factor that multiplies the angle resolution. Used when determining the neighbouring rays. Default 1.2

**radar\_list** [list of Radar objects] Optional. list of radar objects

**trajectory** [Trajectory object] containing trajectory samples

#### Returns

**trajectory** [Trajectory object] Object holding time series

**ind\_rad** [int] radar index

`pyrad.proc.process_traj.process_traj_lightning` (*procstatus, dscfg, radar\_list=None, trajectory=None*)

Return time series according to lightning trajectory

**Parameters**

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
- datatype** [list of string. Dataset keyword] The input data types
  - data\_is\_log** [dict. Dataset keyword] Dictionary specifying for each field if it is in log (True) or linear units (False). Default False
  - ang\_tol** [float. Dataset keyword] Factor that multiplies the angle resolution. Used when determining the neighbouring rays. Default 1.2
- radar\_list** [list of Radar objects] Optional. list of radar objects
- trajectory** [Trajectory object] containing trajectory samples

**Returns**

- trajectory** [Trajectory object] Object holding time series
- ind\_rad** [int] radar index

`pyrad.proc.process_traj.process_traj_trt` (*procstatus*, *dscfg*, *radar\_list=None*, *trajectory=None*)

Processes data according to TRT trajectory

**Parameters**

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:
- datatype** [list of string. Dataset keyword] The input data types
  - time\_tol** [float. Dataset keyword] tolerance between reference time of the radar volume and that of the TRT cell [s]. Default 100.
  - alt\_min, alt\_max** [float. Dataset keyword] Minimum and maximum altitude of the data inside the TRT cell to retrieve [m MSL]. Default None
  - cell\_center** [Bool. Dataset keyword] If True only the range gate closest to the center of the cell is extracted. Default False
  - latlon\_tol** [Float. Dataset keyword] Tolerance in lat/lon when extracting data only from the center of the TRT cell. Default 0.01
- radar\_list** [list of Radar objects] Optional. list of radar objects
- trajectory** [Trajectory object] containing trajectory samples

**Returns**

- new\_dataset** [dictionary] Dictionary containing radar\_out, a radar object containing only data from inside the TRT cell
- ind\_rad** [int] radar index

`pyrad.proc.process_traj.process_traj_trt_contour` (*procstatus*, *dscfg*, *radar\_list=None*, *trajectory=None*)

Gets the TRT cell contour corresponding to each radar volume

**Parameters**

- procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing
- dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**time\_tol** [float. Dataset keyword] tolerance between reference time of the radar volume and that of the TRT cell [s]. Default 100.

**radar\_list** [list of Radar objects] Optional. list of radar objects

**trajectory** [Trajectory object] containing trajectory samples

#### Returns

**new\_dataset** [dict] Dictionary containing radar\_out and roi\_dict. Radar out is the current radar object. roi\_dict contains the positions defining the TRT cell contour

**ind\_rad** [int] radar index

`pyrad.proc.process_traj.process_trajectory` (*procstatus*, *dscfg*, *radar\_list=None*, *trajectory=None*)

Return trajectory

#### Parameters

**procstatus** [int] Processing status: 0 initializing, 1 processing volume, 2 post-processing

**dscfg** [dictionary of dictionaries] data set configuration. Accepted Configuration Keywords:

**datatype** [list of string. Dataset keyword] The input data types

**radar\_list** [list of Radar objects] Optional. list of radar objects

**trajectory** [Trajectory object] containing trajectory samples

#### Returns

**new\_dataset** [Trajectory object] radar object

**ind\_rad** [int] None

## PYRAD.PROD.PRODUCT\_AUX

Auxiliary functions to generate products

---

`get_prodgen_func(dsformat, dsname, dstype)` maps the dataset format into its processing function

---

`pyrad.prod.product_aux.get_prodgen_func(dsformat, dsname, dstype)`  
maps the dataset format into its processing function

### Parameters

**dsformat** [str] dataset group. The following is a list of dataset groups with the function that is called to generate their products. For details about what the functions do check the function documentation:

'VOL':	generate_vol_products	'COLOCATED_GATES':	generate_colocated_gates_products
'COSMO_COORD':	generate_cosmo_coord_products	'GRID':	generate_grid_products
'COSMO2RADAR':	generate_cosmo_to_radar_products	'SPECTRA':	generate_spectra_products
'GRID_TIMEAVG':	generate_grid_time_avg_products	'INTERCOMP':	generate_intercomp_products
'ML':	generate_ml_products	'MONITORING':	generate_monitoring_products
'OC-CURRENCE':	generate_occurrence_products	'QVP':	generate_qvp_products
'SPARSE_GRID':	generate_sparse_grid_products	'SUN_HITS':	generate_sun_hits_products
'TIMEAVG':	generate_time_avg_products	'TIMESERIES':	generate_timeseries_products
'TRAJ_ONLY':	generate_traj_product		

### Returns

**func** [function] pyrad function used to generate the products





## PYRAD.PROD.PROCESS\_PRODUCT

Functions for obtaining Pyrad products from the datasets

<i>generate_occurrence_products</i> (dataset, prdcfg)	generates occurrence products. Accepted product types:
<i>generate_cosmo_coord_products</i> (dataset, prdcfg)	generates COSMO coordinates products. Accepted product types:
<i>generate_cosmo_to_radar_products</i> (dataset, prdcfg)	generates COSMO data in radar coordinates products.
<i>generate_sun_hits_products</i> (dataset, prdcfg)	generates sun hits products. Accepted product types:
<i>generate_qvp_products</i> (dataset, prdcfg)	Generates quasi vertical profile-like products.
<i>generate_ml_products</i> (dataset, prdcfg)	Generates melting layer products. Accepted product types:

`pyrad.prod.process_product.generate_cosmo_coord_products` (*dataset*, *prdcfg*)

**generates COSMO coordinates products. Accepted product types:**

**‘SAVEVOL’:** Save an object containing the index of the COSMO model grid that corresponds to each radar gate in a C/F radial file. User defined parameters:

**file\_type:** **str** The type of file used to save the data. Can be ‘nc’ or ‘h5’. Default ‘nc’

**physical:** **Bool** If True the data will be saved in physical units (floats). Otherwise it will be quantized and saved as binary

**compression:** **str** For ODIM file formats, the type of compression. Can be any of the allowed compression types for hdf5 files. Default gzip

**compression\_opts:** **any** The compression options allowed by the hdf5. Depends on the type of compression. Default 6 (The gzip compression level).

### Parameters

**dataset** [tuple] radar object containing the COSMO coordinates

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

### Returns

**filename** [str] the name of the file created. None otherwise

`pyrad.prod.process_product.generate_cosmo_to_radar_products` (*dataset*, *prdcfg*)

**generates COSMO data in radar coordinates products. Accepted product types:**

**‘SAVEVOL’:** Save an object containing the COSMO data in radar

**coordinates** in a C/F radial or ODIM file. User defined parameters: `file_type`: str

The type of file used to save the data. Can be 'nc' or 'h5'. Default 'nc'

**physical**: **Bool** If True the data will be saved in physical units (floats). Otherwise it will be quantized and saved as binary

**compression**: **str** For ODIM file formats, the type of compression. Can be any of the allowed compression types for hdf5 files. Default gzip

**compression\_opts**: **any** The compression options allowed by the hdf5. Depends on the type of compression. Default 6 (The gzip compression level).

All the products of the 'VOL' dataset group

#### Parameters

**dataset** [tuple] radar object containing the COSMO coordinates

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

#### Returns

**filename** [str] the name of the file created. None otherwise

`pyrad.prod.process_product.generate_ml_products(dataset, prdcfg)`

**Generates melting layer products. Accepted product types:**

**'ML\_TS': Plots and writes a time series of the melting layer, i.e.** the evolution of the average and standard deviation of the melting layer top and thickness and the the number of rays used in the retrieval. User defined parameters:

**dpi**: **int** The pixel density of the plot. Default 72

**'SAVE\_ML': Saves an object containing the melting layer retrieval** information in a C/F radial file

All the products of the 'VOL' dataset group

#### Parameters

**dataset** [dict] dictionary containing the radar object and a keyword stating the status of the processing

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

#### Returns

**filename** [str] the name of the file created. None otherwise

`pyrad.prod.process_product.generate_occurrence_products(dataset, prdcfg)`

**generates occurrence products. Accepted product types:**

**'WRITE\_EXCESS\_GATES': Write the data that identifies radar gates** with clutter that has a frequency of occurrence above a certain threshold. User defined parameters:

**quant\_min**: **float** Minimum frequency of occurrence in percentage to keep the gate as valid. Default 95.

All the products of the 'VOL' dataset group

#### Parameters

**dataset** [tuple] radar object and metadata dictionary

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

#### Returns

**filename** [str] the name of the file created. None otherwise

`pyrad.prod.process_product.generate_qvp_products(dataset, prdcfg)`

Generates quasi vertical profile-like products. Quasi vertical profiles come from azimuthal averaging of polarimetric radar data. With the variable 'qvp\_type' the user decides if the product has to be generated at the end of the processing period ('final') or instantaneously ('instant') Accepted product types:

All the products of the 'VOL' dataset group

#### Parameters

**dataset** [dict] dictionary containing the radar object and a keyword stating the status of the processing

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

#### Returns

**filename** [str] the name of the file created. None otherwise

`pyrad.prod.process_product.generate_sun_hits_products(dataset, prdcfg)`

**generates sun hits products. Accepted product types:**

**'PLOT\_SUN\_HITS': Plots in a sun-radar azimuth difference-sun-radar** elevation difference grid the values of all sun hits obtained during the processing period

**'PLOT\_SUN\_RETRIEVAL': Plots in a sun-radar azimuth difference-sun-** radar elevation difference grid the retrieved sun pattern

**'PLOT\_SUN\_RETRIEVAL\_TS': Plots time series of the retrieved sun** pattern parameters User defined parameters:

**dpi: int** The pixel density of the plot. Default 72

**add\_date\_in\_fname: Bool** If true the year is added in the plot file name

**'WRITE\_SUN\_HITS': Writes the information concerning possible sun hits** in a csv file

**'WRITE\_SUN\_RETRIEVAL': Writes the retrieved sun pattern parameters in** a csv file. User defined parameters:

**add\_date\_in\_fname: Bool** If true the year is added in the csv file name

All the products of the 'VOL' dataset group

#### Parameters

**dataset** [tuple] radar object and sun hits dictionary

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

#### Returns

**filename** [str] the name of the file created. None otherwise



## PYRAD.PROD.PROCESS\_VOL\_PRODUCTS

Functions for obtaining Pyrad products from a radar volume dataset

<i>generate_vol_products</i> (dataset, prdcfg)	Generates radar volume products. Accepted product types:
--	--

---

`pyrad.prod.process_vol_products.generate_vol_products(dataset, prdcfg)`

**Generates radar volume products. Accepted product types:**

**‘CDF’: plots and writes the cumulative density function of data**

**User defined parameters:**

**quantiles: list of floats** The quantiles to compute in percent. Default None

**sector: dict** dictionary defining the sector where to compute the CDF. Default is None and the CDF is computed over all the data May contain:

**rmin, rmax: float** min and max range [m]

**azmin, azmax: float** min and max azimuth angle [deg]

**elmin, elmax: float** min and max elevation angle [deg]

**hmin, hmax: float** min and max altitude [m MSL]

**vismin: float** The minimum visibility to use the data. Default None

**absolute: Bool** If true the absolute values of the data will be used. Default False

**use\_nans: Bool** If true NaN values will be used. Default False

**nan\_value: Bool** The value by which the NaNs are substituted if NaN values are to be used in the computation

**filterclt: Bool** If True the gates containing clutter are filtered

**filterprec: list of ints** The hydrometeor types that are filtered from the analysis. Default empty list.

**‘BSCOPE\_IMAGE’: Creates a B-scope image (azimuth, range)**

**User defined parameters:**

**anglenr [int]** The elevation angle number to use

**ray\_dim [str]** the ray dimension. Can be ‘ang’ or ‘time’. Default ‘ang’

**axis\_rng [bool]** if True the range will be in the x-axis. Otherwise it will be in the y-axis. Default True

**vmin, vmax: float or None** The minimum and maximum values of the color scale. If None the scale is going to be set according to the Py-ART config file

**‘CAPPI\_IMAGE’: Creates a CAPPI image**

**User defined parameters:**

**altitude: flt** CAPPI altitude [m MSL]

**wfunc: str** The function used to produce the CAPPI as defined in `pyart.map.grid_from_radars`. Default ‘NEAREST\_NEIGHBOUR’

**cappi\_res: float** The CAPPI resolution [m]. Default 500.

**‘FIELD\_COVERAGE’: Gets the field coverage over a certain sector**

**User defined parameters:**

**threshold: float or None** Minimum value to consider the data valid. Default None

**nvalid\_min: float** Minimum number of valid gates in the ray to consider it valid. Default 5

**ele\_res, azi\_res: float** Elevation and azimuth resolution of the sectors [deg]. Default 1. and 2.

**ele\_min, ele\_max: float** Min and max elevation angle defining the sector [deg]. Default 0. and 30.

**ele\_step: float** Elevation step [deg]. Default 5.

**ele\_sect\_start, ele\_sect\_stop: float or None** start and stop angles of the sector coverage. Default None

**quantiles: list of floats** The quantiles to compute in the sector. Default 10. to 90. by steps of 10.

**AngTol: float** The tolerance in elevation angle when putting the data in a fixed grid

**‘FIXED\_RNG\_IMAGE’: Plots a fixed range image**

**User defined parameters:**

**AngTol [float]** The tolerance between the nominal angles and the actual radar angles. Default 1.

**ele\_res, azi\_res: float or None** The resolution of the fixed grid [deg]. If None it will be obtained from the separation between angles

**vmin, vmax [float or None]** Min and Max values of the color scale. If None the values are taken from the Py-ART config file

**‘FIXED\_RNG\_SPAN\_IMAGE’: Plots a user-defined statistic over a fixed range image** User defined parameters:

**AngTol [float]** The tolerance between the nominal angles and the actual radar angles. Default 1.

**ele\_res, azi\_res: float or None** The resolution of the fixed grid [deg]. If None it will be obtained from the separation between angles

**stat [str]** The statistic to compute. Can be ‘min’, ‘max’, ‘mean’, ‘mode’. Default ‘max’

**‘HISTOGRAM’: Computes a histogram of the radar volum data**

**User defined parameters:**

**step: float or None** the data quantization step. If none it will be obtained from the Py-ART configuration file

**write\_data: Bool** If true the histogram data is written in a csv file

**‘PLOT\_ALONG\_COORD’:** Plots the radar volume data along a particular coordinate User defined parameters:

**colors:** list of str or None The colors of each plotted line

**mode:** str Plotting mode. Can be ‘ALONG\_RNG’, ‘ALONG\_AZI’ or ‘ALONG\_ELE’

**value\_start, value\_stop:** float The starting and ending points of the data to plot. According to the mode it may refer to the range, azimuth or elevation. If not specified the minimum and maximum possible values are used

**fix\_elevations, fix\_azimuths, fix\_ranges:** list of floats The elevations, azimuths or ranges to plot for each mode. ‘ALONG\_RNG’ would use fix\_elevations and fix\_azimuths ‘ALONG\_AZI’ fix\_ranges and fix\_elevations ‘ALONG\_ELE’ fix\_ranges and fix\_azimuths

**AngTol:** float The tolerance to match the radar angle to the fixed angles Default 1.

**RngTol:** float The tolerance to match the radar range to the fixed ranges Default 50.

**‘PPI\_CONTOUR’:** Plots a PPI countour plot

User defined parameters:

**contour\_values:** list of floats or None The list of contour values to plot. If None the contour values are going to be obtained from the Py-ART config file either with the dictionary key ‘contour\_values’ or from the minimum and maximum values of the field with an assumed division of 10 levels.

**anglenr:** float The elevation angle number

**‘PPI\_CONTOUR\_OVERPLOT’:** Plots a PPI of a field with another field overplotted as a contour plot. User defined parameters:

**contour\_values:** list of floats or None The list of contour values to plot. If None the contour values are going to be obtained from the Py-ART config file either with the dictionary key ‘contour\_values’ or from the minimum and maximum values of the field with an assumed division of 10 levels.

**anglenr:** float The elevation angle number

**‘PPI\_IMAGE’:** Plots a PPI image. It can also plot the histogram and the quantiles of the data in the PPI. User defined parameters:

**anglenr:** float The elevation angle number

**plot\_type:** str The type of plot to perform. Can be ‘PPI’, ‘QUANTILES’ or ‘HISTOGRAM’

**step:** float or None If the plot type is ‘HISTOGRAM’, the width of the histogram bin. If None it will be obtained from the Py-ART config file

**quantiles:** list of float or None If the plot type is ‘QUANTILES’, the list of quantiles to compute. If None a default list of quantiles will be computed

**vmin, vmax:** float or None The minimum and maximum values of the color scale. If None the scale is going to be set according to the Py-ART config file

**‘PPI\_MAP’:** Plots a PPI image over a map. The map resolution and the type of maps used are defined in the variables ‘mapres’ and ‘maps’ in ‘ppiMapImageConfig’ in the loc config file. User defined parameters:

**anglenr:** float The elevation angle number

**‘PPIMAP\_ROI\_OVERPLOT’:** Over plots a polygon delimiting a region of interest on a PPI map. The map resolution and the type of maps used are defined in the variables ‘mapres’ and ‘maps’ in ‘ppiMapImageConfig’ in the loc config file. User defined parameters:

**anglenr: float** The elevation angle number

**‘PROFILE\_STATS’:** Computes and plots a vertical profile statistics. The statistics are saved in a csv file User defined parameters:

**heightResolution: float** The height resolution of the profile [m]. Default 100.

**heightMin, heightMax: float or None** The minimum and maximum altitude of the profile [m MSL]. If None the values will be obtained from the minimum and maximum gate altitude.

**quantity: str** The type of statistics to plot. Can be ‘quantiles’, ‘mode’, ‘regression\_mean’ or ‘mean’.

**quantiles: list of floats** If quantity type is ‘quantiles’ the list of quantiles to compute. Default 25., 50., 75.

**nvalid\_min: int** The minimum number of valid points to consider the statistic valid. Default 4

**make\_linear: Bool** If true the data is converted from log to linear before computing the stats

**include\_nans: Bool** If true NaN values are included in the statistics

**fixed\_span: Bool** If true the profile plot has a fix X-axis

**vmin, vmax: float or None** If fixed\_span is set, the minimum and maximum values of the X-axis. If None, they are obtained from the Py-ART config file

**‘PSEUDOPPI\_CONTOUR’:** Plots a pseudo-PPI countour plot

User defined parameters:

**contour\_values: list of floats or None** The list of contour values to plot. If None the contour values are going to be obtained from the Py-ART config file either with the dictionary key ‘contour\_values’ or from the minimum and maximum values of the field with an assumed division of 10 levels.

**angle: float** The elevation angle at which compute the PPI

**EleTol: float** The tolerance between the actual radar elevation angle and the nominal pseudo-PPI elevation angle.

**‘PSEUDOPPI\_CONTOUR\_OVERPLOT’:** Plots a pseudo-PPI of a field with another field over-plotted as a contour plot User defined parameters:

**contour\_values: list of floats or None** The list of contour values to plot. If None the contour values are going to be obtained from the Py-ART config file either with the dictionary key ‘contour\_values’ or from the minimum and maximum values of the field with an assumed division of 10 levels.

**angle: float** The elevation angle at which compute the PPI

**EleTol: float** The tolerance between the actual radar elevation angle and the nominal pseudo-PPI elevation angle.

**‘PSEUDOPPI\_IMAGE’:** Plots a pseudo-PPI image. It can also plot the histogram and the quantiles of the data in the pseudo-PPI. User defined parameters:

**angle: float** The elevation angle of the pseudo-PPI



**EleTol: float** The tolerance between the actual radar elevation angle and the nominal pseudo-PPI elevation angle.

**plot\_type: str** The type of plot to perform. Can be 'PPI', 'QUANTILES' or 'HISTOGRAM'

**step: float or None** If the plot type is 'HISTOGRAM', the width of the histogram bin. If None it will be obtained from the Py-ART config file

**quantiles: list of float or None** If the plot type is 'QUANTILES', the list of quantiles to compute. If None a default list of quantiles will be computed

**vmin, vmax** [float or None] Min and Max values of the color scale. If None the values are taken from the Py-ART config file

**'PSEUDOPPI\_MAP': Plots a pseudo-PPI image over a map. The map** resolution and the type of maps used are defined in the variables 'mapres' and 'maps' in 'ppiMapImageConfig' in the loc config file. User defined parameters:

**angle: float** The elevation angle of the pseudo-PPI

**EleTol: float** The tolerance between the actual radar elevation angle and the nominal pseudo-PPI elevation angle.

**'PSEUDORHI\_CONTOUR': Plots a pseudo-RHI countour plot**

User defined parameters:

**contour\_values: list of floats or None** The list of contour values to plot. If None the contour values are going to be obtained from the Py-ART config file either with the dictionary key 'contour\_values' or from the minimum and maximum values of the field with an assumed division of 10 levels.

**angle: float** The azimuth angle at which to compute the RPI

**AziTol: float** The tolerance between the actual radar azimuth angle and the nominal pseudo-RHI azimuth angle.

**'PSEUDORHI\_CONTOUR\_OVERPLOT': Plots a pseudo-RHI of a field with** another field over-plotted as a contour plot User defined parameters:

**contour\_values: list of floats or None** The list of contour values to plot. If None the contour values are going to be obtained from the Py-ART config file either with the dictionary key 'contour\_values' or from the minimum and maximum values of the field with an assumed division of 10 levels.

**angle: float** The azimuth angle at which to compute the RPI

**AziTol: float** The tolerance between the actual radar azimuth angle and the nominal pseudo-RHI azimuth angle.

**'PSEUDORHI\_IMAGE': Plots a pseudo-RHI image. It can also plot the** histogram and the quantiles of the data in the pseudo-RHI. User defined parameters:

**angle: float** The azimuth angle at which to compute the RPI

**AziTol: float** The tolerance between the actual radar azimuth angle and the nominal pseudo-RHI azimuth angle.

**plot\_type: str** The type of plot to perform. Can be 'RHI', 'QUANTILES' or 'HISTOGRAM'

**step: float or None** If the plot type is 'HISTOGRAM', the width of the histogram bin. If None it will be obtained from the Py-ART config file

**quantiles: list of float or None** If the plot type is 'QUANTILES', the list of quantiles to compute. If None a default list of quantiles will be computed

**vmin, vmax** [float or None] Min and Max values of the color scale. If None the values are taken from the Py-ART config file

#### **'QUANTILES': Plots and writes the quantiles of a radar volume**

##### **User defined parameters:**

**quantiles: list of floats or None** the list of quantiles to compute. If None a default list of quantiles will be computed.

**write\_data: Bool** If True the computed data will be also written in a csv file

**fixed\_span: Bool** If true the quantile plot has a fix Y-axis

**vmin, vmax: float or None** If fixed\_span is set, the minimum and maximum values of the Y-axis. If None, they are obtained from the Py-ART config file

#### **'RHI\_CONTOUR': Plots an RHI countour plot**

##### **User defined parameters:**

**contour\_values: list of floats or None** The list of contour values to plot. If None the contour values are going to be obtained from the Py-ART config file either with the dictionary key 'contour\_values' or from the minimum and maximum values of the field with an assumed division of 10 levels.

**anglenr: int** The azimuth angle number

#### **'RHI\_CONTOUR\_OVERPLOT': Plots an RHI of a field with another field** over-plotted as a contour plot User defined parameters:

**contour\_values: list of floats or None** The list of contour values to plot. If None the contour values are going to be obtained from the Py-ART config file either with the dictionary key 'contour\_values' or from the minimum and maximum values of the field with an assumed division of 10 levels.

**anglenr: int** The azimuth angle number

#### **'RHI\_IMAGE': Plots an RHI image. It can also plot the** histogram and the quantiles of the data in the RHI. User defined parameters:

**anglenr: int** The azimuth angle number

**plot\_type: str** The type of plot to perform. Can be 'RHI', 'QUANTILES' or 'HISTOGRAM'

**step: float or None** If the plot type is 'HISTOGRAM', the width of the histogram bin. If None it will be obtained from the Py-ART config file

**quantiles: list of float or None** If the plot type is 'QUANTILES', the list of quantiles to compute. If None a default list of quantiles will be computed

**vmin, vmax: float or None** The minimum and maximum values of the color scale. If None the scale is going to be set according to the Py-ART config file

#### **'RHI\_PROFILE': Computes and plots a vertical profile statistics out of** an RHI. The statistics are saved in a csv file User defined parameters:

**rangeStart, rangeStop: float** The range start and stop of the data to extract from the RHI to compute the statistics [m]. Default 0., 25000.

**heightResolution: float** The height resolution of the profile [m]. Default 100.

**heightMin, heightMax: float or None** The minimum and maximum altitude of the profile [m MSL]. If None the values will be obtained from the minimum and maximum gate altitude.

**quantity: str** The type of statistics to plot. Can be 'quantiles', 'mode', 'regression\_mean' or 'mean'.

**quantiles: list of floats** If quantity type is 'quantiles' the list of quantiles to compute. Default 25., 50., 75.

**nvalid\_min: int** The minimum number of valid points to consider the statistic valid. Default 4

**make\_linear: Bool** If true the data is converted from log to linear before computing the stats

**include\_nans: Bool** If true NaN values are included in the statistics

**fixed\_span: Bool** If true the profile plot has a fix X-axis

**vmin, vmax: float or None** If fixed\_span is set, the minimum and maximum values of the X-axis. If None, they are obtained from the Py-ART config file

**'SAVEALL': Saves radar volume data including all or a list of user-** defined fields in a C/F radial or ODIM file User defined parameters:

**file\_type: str** The type of file used to save the data. Can be 'nc' or 'h5'. Default 'nc'

**datatypes: list of str or None** The list of data types to save. If it is None, all fields in the radar object will be saved

**physical: Bool** If True the data will be saved in physical units (floats). Otherwise it will be quantized and saved as binary

**compression: str** For ODIM file formats, the type of compression. Can be any of the allowed compression types for hdf5 files. Default gzip

**compression\_opts: any** The compression options allowed by the hdf5. Depends on the type of compression. Default 6 (The gzip compression level).

**'SAVESTATE': Saves the last processed data in a file. Used for real-** time data processing

**'SAVEVOL': Saves one field of a radar volume data in a C/F radial or** ODIM file User defined parameters:

**file\_type: str** The type of file used to save the data. Can be 'nc' or 'h5'. Default 'nc'

**physical: Bool** If True the data will be saved in physical units (floats). Otherwise it will be quantized and saved as binary

**compression: str** For ODIM file formats, the type of compression. Can be any of the allowed compression types for hdf5 files. Default gzip

**compression\_opts: any** The compression options allowed by the hdf5. Depends on the type of compression. Default 6 (The gzip compression level).

**'SAVE\_FIXED\_ANGLE': Saves the position of the first fix angle in a** csv file

**'SELFCONSISTENCY': Plots a ZDR versus KDP/ZH histogram of data.**

User defined parameters:

**retrieve\_relation** [bool] If True plots also the retrieved relationship. Default True

**plot\_theoretical** [bool] If True plots also the theoretical relationship. Default True

**normalize** [bool] If True the occurrence density of ZK/KDP for each ZDR bin is going to be represented. Otherwise it will show the number of gates at each bin. Default True

**‘TIME\_RANGE’: Plots a time-range plot**

**User defined parameters:**

**anglenr: float** The number of the fixed angle to plot

**‘WIND\_PROFILE’: Plots vertical profile of wind data (U, V, W components and wind velocity and direction) out of a radar volume containing the retrieved U,V and W components of the wind, the standard deviation of the retrieval and the velocity difference between the estimated radial velocity (assuming the wind to be uniform) and the actual measured radial velocity. User defined parameters:**

**heightResolution: float** The height resolution of the profile [m]. Default 100.

**heightMin, heightMax: float or None** The minimum and maximum altitude of the profile [m MSL]. If None the values will be obtained from the minimum and maximum gate altitude.

**min\_ele: float** The minimum elevation to be used in the computation of the vertical velocities. Default 5.

**max\_ele: float** The maximum elevation to be used in the computation of the horizontal velocities. Default 85.

**fixed\_span: Bool** If true the profile plot has a fix X-axis

**vmin, vmax: float or None** If fixed\_span is set, the minimum and maximum values of the X-axis. If None, they are obtained from the span of the U component defined in the Py-ART config file

**Parameters**

**dataset** [dict] dictionary with key radar\_out containing a radar object

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

**Returns**

**The list of created fields or None**

## PYRAD.PROD.PROCESS\_GRID\_PRODUCTS

Functions for obtaining Pyrad products from gridded datasets

<i>generate_grid_time_avg_products</i> (dataset, prdcfg)	generates time average products. Accepted product types:
<i>generate_sparse_grid_products</i> (dataset, prdcfg)	generates products defined by sparse points. Accepted product types:
<i>generate_grid_products</i> (dataset, prdcfg)	generates grid products. Accepted product types:

`pyrad.prod.process_grid_products.generate_grid_products (dataset, prdcfg)`

**generates grid products. Accepted product types:**

**‘CROSS\_SECTION’: Plots a cross-section of gridded data**

**User defined parameters:**

**coord1, coord2: dict** The two lat-lon coordinates marking the limits. They have the keywords ‘lat’ and ‘lon’ [degree]. The altitude limits are defined by the parameters in ‘rhiImageConfig’ in the ‘loc’ configuration file

**‘HISTOGRAM’: Computes a histogram of the radar volum data**

**User defined parameters:**

**step: float or None** the data quantization step. If none it will be obtained from the Py-ART configuration file

**write\_data: Bool** If true the histogram data is written in a csv file

**‘LATITUDE\_SLICE’: Plots a cross-section of gridded data over a constant latitude.** User defined parameters:

**lon, lat: floats** The starting point of the cross-section. The ending point is defined by the parameters in ‘rhiImageConfig’ in the ‘loc’ configuration file

**‘LONGITUDE\_SLICE’: Plots a cross-ection of gridded data over a constant longitude.** User defined parameters:

**lon, lat: floats** The starting point of the cross-section. The ending point is defined by the parameters in ‘rhiImageConfig’ in the ‘loc’ configuration file

**‘SAVEALL’: Saves a gridded data object including all or a list of** user-defined fields in a netcdf file  
User defined parameters:

**datatypes: list of str or None** The list of data types to save. If it is None, all fields in the radar object will be saved

‘SAVEVOL’: Saves on field of a gridded data object in a netcdf file. ‘SURFACE\_IMAGE’: Plots a surface image of gridded data.

**User defined parameters:**

**level: int** The altitude level to plot. The rest of the parameters are defined by the parameters in ‘ppiImageConfig’ and ‘ppiMapImageConfig’ in the ‘loc’ configuration file

**‘SURFACE\_CONTOUR’: Plots a surface image of gridded data.**

**User defined parameters:**

**level: int** The altitude level to plot. The rest of the parameters are defined by the parameters in ‘ppiImageConfig’ and ‘ppiMapImageConfig’ in the ‘loc’ configuration file

**Parameters**

**dataset** [grid] grid object

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

**Returns**

**None or name of generated files**

`pyrad.prod.process_grid_products.generate_grid_time_avg_products(dataset, prdcfg)`

**generates time average products. Accepted product types:** All the products of the ‘VOL’ dataset group

**Parameters**

**dataset** [tuple] radar objects and colocated gates dictionary

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

**Returns**

**filename** [str] the name of the file created. None otherwise

`pyrad.prod.process_grid_products.generate_sparse_grid_products(dataset, prdcfg)`

**generates products defined by sparse points. Accepted product types:**

**‘SURFACE\_IMAGE’: Generates a surface image**

**User defined parameters:**

**‘field\_limits’: list of floats** The limits of the surface to plot [deg] lon0, lon1, lat0, lat1

**Parameters**

**dataset** [dictionary containing the points and their values]

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

**Returns**

**no return**

## PYRAD.PROD.PROCESS\_TIMESERIES\_PRODUCTS

Functions for obtaining Pyrad products from a time series datasets

`generate_timeseries_products(dataset, prd- Generates time series products. Accepted product types:  
cfg)`

---

`pyrad.prod.process_timeseries_products.generate_timeseries_products(dataset,  
prdcfg)`

**Generates time series products. Accepted product types:**

**‘COMPARE\_CUMULATIVE\_POINT’:** Plots in the same graph 2 time series of data accumulation (typically rainfall rate). One time series is a point measurement of radar data while the other is from a co-located instrument (rain gauge or disdrometer) User defined parameters:

**dpi: int** The pixel density of the plot. Default 72

**vmin, vmax: float** The limits of the Y-axis. If none they will be obtained from the Py-ART config file.

**sensor: str** The sensor type. Can be ‘rgage’ or ‘disdro’

**sensorid: str** The sensor ID.

**location: str** A string identifying the location of the disdrometer

**freq: float** The frequency used to retrieve the polarimetric variables of a disdrometer

**ele: float** The elevation angle used to retrieve the polarimetric variables of a disdrometer

**ScanPeriod: float** The scanning period of the radar in seconds. This parameter is defined in the ‘loc’ config file

**‘COMPARE\_POINT’:** Plots in the same graph 2 time series of data . One time series is a point measurement of radar data while the other is from a co-located instrument (rain gauge or disdrometer) User defined parameters:

**dpi: int** The pixel density of the plot. Default 72

**vmin, vmax: float** The limits of the Y-axis. If none they will be obtained from the Py-ART config file.

**sensor: str** The sensor type. Can be ‘rgage’ or ‘disdro’

**sensorid: str** The sensor ID.

**location: str** A string identifying the location of the disdrometer

**freq: float** The frequency used to retrieve the polarimetric variables of a disdrometer

**ele: float** The elevation angle used to retrieve the polarimetric variables of a disdrometer

**‘COMPARE\_TIME\_AVG’: Creates a scatter plot of average radar data** versus average sensor data.  
User defined parameters:

**dpi: int** The pixel density of the plot. Default 72

**sensor: str** The sensor type. Can be ‘rgage’ or ‘disdro’

**sensorid: str** The sensor ID.

**location: str** A string identifying the location of the disdrometer

**freq: float** The frequency used to retrieve the polarimetric variables of a disdrometer

**ele: float** The elevation angle used to retrieve the polarimetric variables of a disdrometer

**cum\_time: float** Data accumulation time [s]. Default 3600.

**base\_time: float** Starting moment of the accumulation [s from midnight]. Default 0.

**‘PLOT\_AND\_WRITE’: Writes and plots a trajectory time series.**

User defined parameters:

**ymin, ymax: float** The minimum and maximum value of the Y-axis. If none it will be obtained from the Py-ART config file.

**‘PLOT\_AND\_WRITE\_POINT’: Plots and writes a time series of radar data** at a particular point  
User defined parameters:

**dpi: int** The pixel density of the plot. Default 72

**vmin, vmax: float** The limits of the Y-axis. If none they will be obtained from the Py-ART config file.

**‘PLOT\_CUMULATIVE\_POINT’: Plots a time series of radar data** accumulation at a particular point. User defined parameters:

**dpi: int** The pixel density of the plot. Default 72

**vmin, vmax: float** The limits of the Y-axis. If none they will be obtained from the Py-ART config file.

**ScanPeriod: float** The scanning period of the radar in seconds. This parameter is defined in the ‘loc’ config file

**‘PLOT\_HIST’: plots and writes a histogram of all the data gathered** during the trajectory processing  
User defined parameters:

**step: float or None** The quantization step of the data. If None it will be obtained from the Py-ART config file

**‘TRAJ\_CAPPI\_IMAGE’: Creates a CAPPI image with the trajectory position** overplot on it. User defined parameters:

**color\_ref: str** The meaning of the color code with which the trajectory is plotted. Can be ‘None’, ‘altitude’ (the absolute altitude), ‘rel\_altitude’ (altitude relative to the CAPPI altitude), ‘time’ (trajectory time respect of the start of the radar scan leading to the CAPPI)

**altitude: float** The CAPPI altitude [m]

**wfunc: str** Function used in the gridding of the radar data. The function types are defined in `pyart.map.grid_from_radars`. Default ‘NEAREST\_NEIGHBOUR’

**res: float** The CAPPI resolution [m]. Default 500.



### Parameters

**dataset** [dictionary] radar object

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

### Returns

**no return**



## PYRAD.PROD.PROCESS\_MONITORING\_PRODUCTS

Functions for obtaining Pyrad products from monitoring datasets

`generate_monitoring_products(dataset, prdcfg)` generates a monitoring product.

---

`pyrad.prod.process_monitoring_products.generate_monitoring_products(dataset, prdcfg)`  
generates a monitoring product. With the parameter 'hist\_type' the user may define if the product is computed for each radar volume ('instant') or at the end of the processing period ('cumulative'). Default is 'cumulative'. Accepted product types:

**'ANGULAR\_DENSITY':** For a specified elevation angle, plots a 2D histogram with the azimuth angle in the X-axis and the data values in the Y-axis. The reference values and the user defined quantiles are also plot on the same figure User defined parameters:

**anglenr: int** The elevation angle number to plot

**quantiles: list of floats** The quantiles to plot. Default 25., 50., 75.

**ref\_value: float** The reference value

**vmin, vmax** [floats or None] The minimum and maximum values of the data points.  
If not specified they are obtained from the Py-ART config file

**'CUMUL\_VOL\_TS':** Plots time series of the average of instantaneous quantiles stored in a csv file. User defined parameters:

**quantiles: list of 3 floats** the quantiles to compute. Default 25., 50., 75.

**ref\_value: float** The reference value. Default 0

**sort\_by\_date: Bool** If true when reading the csv file containing the statistics the data is sorted by date. Default False

**rewrite: Bool** If true the csv file containing the statistics is rewritten

**add\_data\_in\_fname: Bool** If true and the data used is cumulative the year is written in the csv file name and the plot file name

**npoints\_min: int** Minimum number of points to use the data point in the plotting and to send an alarm. Default 0

**vmin, vmax: float or None** Limits of the Y-axis (data value). If None the limits are obtained from the Py-ART config file

**alarm: Bool** If true an alarm is sent

**tol\_abs: float** Margin of tolerance from the reference value. If the current value is above this margin an alarm is sent. If the margin is not specified it is not possible to send any alarm

**tol\_trend: float** Margin of tolerance from the reference value. If the trend of the last X events is above this margin an alarm is sent. If the margin is not specified it is not possible to send any alarm

**nevents\_min: int** Minimum number of events with sufficient points to send an alarm related to the trend. If not specified it is not possible to send any alarm

**sender: str** The mail of the alarm sender. If not specified it is not possible to send any alarm

**receiver\_list: list of str** The list of emails of the people that will receive the alarm.. If not specified it is not possible to send any alarm

**‘PPI\_HISTOGRAM’: Plots a histogram of data at a particular** elevation angle. User defined parameters:

**anglenr: int** The elevation angle number to plot

**‘SAVEVOL’: Saves the monitoring data in a C/F radar file. The data** field contains histograms of data for each pair of azimuth and elevation angles

**‘VOL\_HISTOGRAM’: Plots a histogram of data collected from all the** radar volume. User defined parameters:

**write\_data: bool** If true the resultant histogram is also saved in a csv file. Default True.

**‘VOL\_TS’: Computes statistics of the gathered data and writes them in** a csv file and plots a time series of those statistics. User defined parameters:

**quantiles: list of 3 floats** the quantiles to compute. Default 25., 50., 75.

**ref\_value: float** The reference value. Default 0

**sort\_by\_date: Bool** If true when reading the csv file containing the statistics the data is sorted by date. Default False

**rewrite: Bool** If true the csv file containing the statistics is rewritten

**add\_data\_in\_fname: Bool** If true and the data used is cumulative the year is written in the csv file name and the plot file name

**npoints\_min: int** Minimum number of points to use the data point in the plotting and to send an alarm. Default 0

**vmin, vmax: float or None** Limits of the Y-axis (data value). If None the limits are obtained from the Py-ART config file

**alarm: Bool** If true an alarm is sent

**tol\_abs: float** Margin of tolerance from the reference value. If the current value is above this margin an alarm is sent. If the margin is not specified it is not possible to send any alarm

**tol\_trend: float** Margin of tolerance from the reference value. If the trend of the last X events is above this margin an alarm is sent. If the margin is not specified it is not possible to send any alarm

**nevents\_min: int** Minimum number of events with sufficient points to send an alarm related to the trend. If not specified it is not possible to send any alarm

**sender: str** The mail of the alarm sender. If not specified it is not possible to send any alarm

**receiver\_list: list of str** The list of emails of the people that will receive the alarm.. If not specified it is not possible to send any alarm

#### Parameters

**dataset** [dictionary] dictionary containing a histogram object and some metadata

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

#### Returns

**filename** [str] the name of the file created. None otherwise



## PYRAD.PROD.PROCESS\_INTERCOMP\_PRODUCTS

Functions for obtaining Pyrad products from datasets used in the intercomparison process

<i>generate_intercomp_products</i> (dataset, prd-cfg)	Generates radar intercomparison products. Accepted product types:
<i>generate_colocated_gates_products</i> (dataset, ...)	Generates colocated gates products. Accepted product types:
<i>generate_time_avg_products</i> (dataset, prdcfg)	generates time average products. Accepted product types:

`pyrad.prod.process_intercomp_products.generate_colocated_gates_products` (*dataset*,  
*prd-cfg*)

**Generates colocated gates products. Accepted product types:**

**‘WRITE\_COLOCATED\_GATES’:** Writes the position of the co-located gates in a csv file

All the products of the ‘VOL’ dataset group

### Parameters

**dataset** [tuple] radar objects and colocated gates dictionary

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

### Returns

**filename** [str] the name of the file created. None otherwise

`pyrad.prod.process_intercomp_products.generate_intercomp_products` (*dataset*,  
*prdcfg*)

**Generates radar intercomparison products. Accepted product types:**

**‘PLOT\_AND\_WRITE\_INTERCOMP\_TS’:** Writes statistics of radar intercomparison in a file and plots the time series of the statistics. User defined parameters:

**‘add\_date\_in\_fname’:** Bool If true adds the year in the csv file containing the statistics.  
Default False

**‘sort\_by\_date’:** Bool If true sorts the statistics by date when reading the csv file containing the statistics. Default False

**‘rewrite’:** Bool If true rewrites the csv file containing the statistics. Default False

**‘npoints\_min’:** int The minimum number of points to consider the statistics valid and therefore use the data point in the plotting. Default 0

**‘corr\_min’: float** The minimum correlation to consider the statistics valid and therefore use the data point in the plotting. Default 0.

**‘PLOT\_SCATTER\_INTERCOMP’: Plots a density plot with the points of** radar 1 versus the points of radar 2 User defined parameters:

**‘step’: float** The quantization step of the data. If none it will be computed using the Py-ART config file. Default None

**‘scatter\_type’: str** Type of scatter plot. Can be a plot for each radar volume (‘instant’) or at the end of the processing period (‘cumulative’). Default is ‘cumulative’

**‘WRITE\_INTERCOMP’: Writes the instantaneously intercompared data** (gate positions, values, etc.) in a csv file.

**‘WRITE\_INTERCOMP\_TIME\_AVG’: Writes the time-averaged intercompared** data (gate positions, values, etc.) in a csv file.

#### Parameters

**dataset** [tuple] values of colocated gates dictionary

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

#### Returns

**filename** [str] the name of the file created. None otherwise

`pyrad.prod.process_intercomp_products.generate_time_avg_products(dataset, prdcfg)`

**generates time average products. Accepted product types:** All the products of the ‘VOL’ dataset group

#### Parameters

**dataset** [tuple] radar objects and colocated gates dictionary

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

#### Returns

**filename** [str] the name of the file created. None otherwise



## PYRAD.PROD.PROCESS\_SPECTRA\_PRODUCTS

Functions for obtaining Pyrad products from spectra datasets

*generate\_spectra\_products*(dataset, prdcfg)      generates spectra products. Accepted product types:

---

`pyrad.prod.process_spectra_products.generate_spectra_products` (*dataset, prdcfg*)

**generates spectra products. Accepted product types:**

**‘AMPLITUDE\_PHASE\_ANGLE\_DOPPLER’:** Makes an angle Doppler plot of complex spectra or IQ data. The plot can be along azimuth or along range. It is plotted separately the module and the phase of the signal. User defined parameters:

**along\_azi** [bool] If true the plot is performed along azimuth, otherwise along elevation.  
Default true

**ang** [float] The fixed angle (deg). Default 0.

**rng** [float] The fixed range (m). Default 0.

**ang\_tol** [float] The fixed angle tolerance (deg). Default 1.

**rng\_tol** [float] The fixed range tolerance (m). Default 50.

**xaxis\_info** [str] The xaxis type. Can be ‘Doppler\_velocity’, ‘Doppler\_frequency’ or ‘pulse\_number’

**ampli\_vmin, ampli\_vmax, phase\_vmin, phase\_vmax** [float or None] Minimum and maximum of the color scale for the module and phase

**‘AMPLITUDE\_PHASE\_DOPPLER’:** Plots a complex Doppler spectrum or IQ data making two separate plots for the module and phase of the signal User defined parameters:

**azi, ele, rng** [float] azimuth and elevation (deg) and range (m) of the ray to plot

**azi\_to, ele\_tol, rng\_tol** [float] azimuth and elevation (deg) and range (m) tolerance respect to nominal position to plot. Default 1, 1, 50.

**ind\_ray, ind\_rng** [int] index of the ray and range to plot. Alternative to defining its antenna coordinates

**xaxis\_info** [str] The xaxis type. Can be ‘Doppler\_velocity’, ‘Doppler\_frequency’ or ‘pulse\_number’

**ampli\_vmin, ampli\_vmax, phase\_vmin, phase\_vmax** [float or None] Minimum and maximum of the color scale for the module and phase

**‘AMPLITUDE\_PHASE\_RANGE\_DOPPLER’:** Plots a complex spectra or IQ data range-Doppler making two separate plots for the module and phase of the signal User defined parameters:

**azi, ele** [float] azimuth and elevation (deg) of the ray to plot

**azi\_to, ele\_tol** [float] azimuth and elevation (deg) tolerance respect to nominal position to plot. Default 1, 1.

**ind\_ray** [int] index of the ray to plot. Alternative to defining its antenna coordinates

**xaxis\_info** [str] The xaxis type. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**ampli\_vmin, ampli\_vmax, phase\_vmin, phase\_vmax** [float or None] Minimum and maximum of the color scale for the module and phase

**'AMPLITUDE\_PHASE\_TIME\_DOPPLER': Plots a complex spectra or IQ data** time-Doppler making two separate plots for the module and phase of the signal User defined parameters:

**xaxis\_info** [str] The xaxis type. Can be 'Doppler\_velocity' or 'Doppler frequency'

**ampli\_vmin, ampli\_vmax, phase\_vmin, phase\_vmax** [float or None] Minimum and maximum of the color scale for the module and phase

**plot\_type** [str] Can be 'final' or 'temporal'. If final the data is only plotted at the end of the processing

**'ANGLE\_DOPPLER': Makes an angle Doppler plot. The plot can be along** azimuth or along range User defined parameters:

**along\_azi** [bool] If true the plot is performed along azimuth, otherwise along elevation. Default true

**ang** [float] The fixed angle (deg). Default 0.

**rng** [float] The fixed range (m). Default 0.

**ang\_tol** [float] The fixed angle tolerance (deg). Default 1.

**rng\_tol** [float] The fixed range tolerance (m). Default 50.

**xaxis\_info** [str] The xaxis type. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**vmin, vmax** [float or None] Minimum and maximum of the color scale

**'COMPLEX\_ANGLE\_DOPPLER': Makes an angle Doppler plot of complex** spectra or IQ data. The plot can be along azimuth or along range. The real and imaginary parts are plotted separately User defined parameters:

**along\_azi** [bool] If true the plot is performed along azimuth, otherwise along elevation. Default true

**ang** [float] The fixed angle (deg). Default 0.

**rng** [float] The fixed range (m). Default 0.

**ang\_tol** [float] The fixed angle tolerance (deg). Default 1.

**rng\_tol** [float] The fixed range tolerance (m). Default 50.

**xaxis\_info** [str] The xaxis type. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**vmin, vmax** [float or None] Minimum and maximum of the color scale

**'COMPLEX\_DOPPLER': Plots a complex Doppler spectrum or IQ data making** two separate plots for the real and imaginary parts User defined parameters:

**azi, ele, rng** [float] azimuth and elevation (deg) and range (m) of the ray to plot

**azi\_to, ele\_tol, rng\_tol** [float] azimuth and elevation (deg) and range (m) tolerance respect to nominal position to plot. Default 1, 1, 50.

**ind\_ray, ind\_rng** [int] index of the ray and range to plot. Alternative to defining its antenna coordinates

**xaxis\_info** [str] The xaxis type. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**vmin, vmax** [float or None] Minimum and maximum of the color scale

**'COMPLEX\_RANGE\_DOPPLER': Plots the complex spectra or IQ data** range-Doppler making two separate plots for the real and imaginary parts User defined parameters:

**azi, ele** [float] azimuth and elevation (deg) of the ray to plot

**azi\_to, ele\_tol** [float] azimuth and elevation (deg) tolerance respect to nominal position to plot. Default 1, 1.

**ind\_ray** [int] index of the ray to plot. Alternative to defining its antenna coordinates

**xaxis\_info** [str] The xaxis type. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**vmin, vmax** [float or None] Minimum and maximum of the color scale

**'COMPLEX\_TIME\_DOPPLER': Plots the complex spectra or IQ data** time-Doppler making two separate plots for the real and imaginary parts User defined parameters:

**xaxis\_info** [str] The xaxis type. Can be 'Doppler\_velocity' or 'Doppler frequency'

**vmin, vmax** [float or None] Minimum and maximum of the color scale

**plot\_type** [str] Can be 'final' or 'temporal'. If final the data is only plotted at the end of the processing

**'DOPPLER': Plots a Doppler spectrum variable or IQ data variable**

User defined parameters:

**azi, ele, rng** [float] azimuth and elevation (deg) and range (m) of the ray to plot

**azi\_to, ele\_tol, rng\_tol** [float] azimuth and elevation (deg) and range (m) tolerance respect to nominal position to plot. Default 1, 1, 50.

**ind\_ray, ind\_rng** [int] index of the ray and range to plot. Alternative to defining its antenna coordinates

**xaxis\_info** [str] The xaxis type. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**vmin, vmax** [float or None] Minimum and maximum of the color scale

**'RANGE\_DOPPLER': Makes a range-Doppler plot of spectral or IQ data**

User defined parameters:

**azi, ele** [float] azimuth and elevation (deg) of the ray to plot

**azi\_to, ele\_tol** [float] azimuth and elevation (deg) tolerance respect to nominal position to plot. Default 1, 1.

**ind\_ray** [int] index of the ray to plot. Alternative to defining its antenna coordinates

**xaxis\_info** [str] The xaxis type. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**vmin, vmax** [float or None] Minimum and maximum of the color scale

**‘SAVEALL’**: Saves radar spectra or IQ volume data including all or a list of userdefined fields in a netcdf file User defined parameters:

**datatypes: list of str or None** The list of data types to save. If it is None, all fields in the radar object will be saved

**physical: Bool** If True the data will be saved in physical units (floats). Otherwise it will be quantized and saved as binary

**‘SAVEVOL’**: Saves one field of a radar spectra or IQ volume data in a netcdf file User defined parameters:

**physical: Bool** If True the data will be saved in physical units (floats). Otherwise it will be quantized and saved as binary

**‘TIME\_DOPPLER’**: Makes a time-Doppler plot of spectral or IQ data at a point of interest. User defined parameters:

**xaxis\_info** [str] The xaxis type. Can be ‘Doppler\_velocity’, ‘Doppler\_frequency’ or ‘pulse\_number’

**vmin, vmax** [float or None] Minimum and maximum of the color scale

**plot\_type** [str] Can be ‘final’ or ‘temporal’. If final the data is only plotted at the end of the processing

#### Parameters

**dataset** [spectra] spectra object

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

#### Returns

None or name of generated files

## PYRAD.PROD.PROCESS\_PRODUCT

Functions for obtaining Pyrad products from the datasets

---

*generate\_traj\_product*(traj, prdcfg) Generates trajectory products. Accepted product types:

---

`pyrad.prod.process_traj_products.generate_traj_product` (*traj*, *prdcfg*)

**Generates trajectory products. Accepted product types:**

**‘TRAJ\_MAP’:** Plots the trajectory on a lat-lon map with the altitude color coded

**‘TRAJ\_PLOT’:** Plots time series of the trajectory respect to the radar elevation, azimuth or range  
User defined parameters:

**‘datatype’:** str The type of parameter: ‘EL’, ‘AZ’, or ‘RANGE’

**‘TRAJ\_TEXT’:** Writes the trajectory information in a csv file

**Parameters**

**traj** [Trajectory object]

**prdcfg** [dictionary of dictionaries] product configuration dictionary of dictionaries

**Returns**

None

---



## PYRAD.IO.IO\_AUX

Auxiliary functions for reading/writing files

<i>get_rad4alp_prod_fname</i> (datatype)	Given a datatype find the corresponding start and termination of the METRANET product file name
<i>map_hydro</i> (hydro_data_op)	maps the operational hydrometeor classification identifiers to the ones used by Py-ART
<i>map_Doppler</i> (Doppler_data_bin, Nyquist_vel)	maps the binary METRANET Doppler data to actual Doppler velocity
<i>get_save_dir</i> (basepath, procname, dsname, prd-name)	obtains the path to a product directory and eventually creates it
<i>make_filename</i> (prdtype, dstype, dsname, ext_list)	creates a product file name
<i>generate_field_name_str</i> (datatype)	Generates a field name in a nice to read format.
<i>get_datatype_metrانet</i> (datatype)	maps the config file radar data type name into the corresponding metranet data type name and Py-ART field name
<i>get_datatype_odim</i> (datatype)	maps the config file radar data type name into the corresponding odim data type name and Py-ART field name
<i>get_fieldname_pyart</i> (datatype)	maps the config file radar data type name into the corresponding rainbow Py-ART field name
<i>get_fieldname_cosmo</i> (field_name)	maps the Py-ART field name into the corresponding COSMO variable name
<i>get_field_unit</i> (datatype)	Return unit of datatype.
<i>get_field_name</i> (datatype)	Return long name of datatype.
<i>get_file_list</i> (datadescriptor, starttimes, ...)	gets the list of files with a time period
<i>get_rad4alp_dir</i> (basepath, voltime[, ...])	gets the directory where rad4alp data is stored
<i>get_rad4alp_grid_dir</i> (basepath, voltime, ...)	gets the directory where rad4alp grid data is stored
<i>get_trtfile_list</i> (basepath, starttime, endtime)	gets the list of TRT files with a time period
<i>get_scan_list</i> (scandescrptor_list)	determine which is the scan list for each radar
<i>get_new_rainbow_file_name</i> (master_fname, ...)	get the rainbow file name containing datatype from a master file name and data type
<i>get_datatype_fields</i> (datadescriptor)	splits the data type descriptor and provides each individual member
<i>get_dataset_fields</i> (datasetdescr)	splits the dataset type descriptor and provides each individual member
<i>get_datetime</i> (fname, datadescriptor)	Given a data descriptor gets date and time from file name
<i>find_raw_cosmo_file</i> (voltime, datatype, cfg)	Search a COSMO file in netcdf format
<i>find_cosmo_file</i> (voltime, datatype, cfg, scanid)	Search a COSMO file in Rainbow format
<i>find_hzt_file</i> (voltime, cfg[, ind_rad])	Search an ISO-0 degree file in HZT format

Continued on next page

Table 1 – continued from previous page

<code>find_rad4alpcosmo_file(voltime, datatype, ...)</code>	Search a COSMO file
<code>find_pyradcosmo_file(basepath, voltime, ...)</code>	Search a COSMO file in CFRadial or ODIM format
<code>_get_datetime(fname, datagroup[, ftime_format])</code>	Given a data group gets date and time from file name
<code>find_date_in_file_name(filename[, date_format])</code>	Find a date with date format defined in date_format in a file name.

`pyrad.io.io_aux._get_datetime(fname, datagroup, ftime_format=None)`

Given a data group gets date and time from file name

#### Parameters

**fname** [str] file name

**datadescriptor** [str] radar field type. Format : [radar file type]:[datatype]

**ftime\_format** [str or None] if the file is of type ODIM this contain the file time format

#### Returns

**fdatetime** [datetime object] date and time in file name

`pyrad.io.io_aux.find_cosmo_file(voltime, datatype, cfg, scanid, ind_rad=0)`

Search a COSMO file in Rainbow format

#### Parameters

**voltime** [datetime object] volume scan time

**datatype** [str] type of COSMO data to look for

**cfg** [dictionary of dictionaries] configuration info to figure out where the data is

**scanid** [str] name of the scan

**ind\_rad** [int] radar index

#### Returns

**fname** [str] Name of COSMO file if it exists. None otherwise

`pyrad.io.io_aux.find_date_in_file_name(filename, date_format='%Y%m%d%H%M%S')`

Find a date with date format defined in date\_format in a file name. If no date is found returns None

#### Parameters

**filename** [str] file name

**date\_format** [str] The time format

#### Returns

**fdatetime** [datetime object] date and time in file name

`pyrad.io.io_aux.find_hzt_file(voltime, cfg, ind_rad=0)`

Search an ISO-0 degree file in HZT format

#### Parameters

**voltime** [datetime object] volume scan time

**cfg** [dictionary of dictionaries] configuration info to figure out where the data is

**ind\_rad** [int] radar index

#### Returns



**fname** [str] Name of HZT file if it exists. None otherwise

`pyrad.io.io_aux.find_pyradcosmo_file(basepath, voltime, datatype, cfg, dataset)`  
Search a COSMO file in CFRadial or ODIM format

#### Parameters

**basepath** [str] base path to the COSMO file

**voltime** [datetime object] volume scan time

**datatype** [str] type of COSMO data to look for

**cfg** [dictionary of dictionaries] configuration info to figure out where the data is

**dataset** [str] name of the folder where the data is stored

#### Returns

**fname** [str] Name of COSMO file if it exists. None otherwise

`pyrad.io.io_aux.find_rad4alpcosmo_file(voltime, datatype, cfg, scanid, ind_rad=0)`  
Search a COSMO file

#### Parameters

**voltime** [datetime object] volume scan time

**datatype** [str] type of COSMO data to look for

**cfg: dictionary of dictionaries** configuration info to figure out where the data is

**ind\_rad: int** radar index

#### Returns

**fname** [str] Name of COSMO file if it exists. None otherwise

**scanid: str** name of the scan

`pyrad.io.io_aux.find_raw_cosmo_file(voltime, datatype, cfg, ind_rad=0)`  
Search a COSMO file in netcdf format

#### Parameters

**voltime** [datetime object] volume scan time

**datatype** [str] type of COSMO data to look for

**cfg** [dictionary of dictionaries] configuration info to figure out where the data is

**ind\_rad** [int] radar index

#### Returns

**fname** [str] Name of COSMO file if it exists. None otherwise

`pyrad.io.io_aux.generate_field_name_str(datatype)`  
Generates a field name in a nice to read format.

#### Parameters

**datatype** [str] The data type

#### Returns

**field\_str** [str] The field name

`pyrad.io.io_aux.get_dataset_fields(datasetdescr)`  
splits the dataset type descriptor and provides each individual member

#### Parameters

**datasetdescr** [str] dataset type. Format : [processing level]:[dataset type]

#### Returns

**proclevel** [str] dataset processing level

**dataset** [str] dataset type, i.e. dBZ, ZDR, ISO0, ...

`pyrad.io.io_aux.get_datatype_fields(datadescriptor)`  
splits the data type descriptor and provides each individual member

#### Parameters

**datadescriptor** [str] radar field type. Format : [radar file type]:[datatype]

#### Returns

**radarnr** [str] radar number, i.e. RADAR1, RADAR2, ...

**datagroup** [str] data type group, i.e. RAINBOW, RAD4ALP, ODIM, CFRADIAL, COSMO, MXPOL ...

**datatype** [str] data type, i.e. dBZ, ZDR, ISO0, ...

**dataset** [str] dataset type (for saved data only)

**product** [str] product type (for saved data only)

`pyrad.io.io_aux.get_datatype_metranet(datatype)`  
maps de config file radar data type name into the corresponding metranet data type name and Py-ART field name

#### Parameters

**datatype** [str] config file radar data type name

#### Returns

**metranet type** [dict] dictionary containing the metranet data type name and its corresponding Py-ART field name

`pyrad.io.io_aux.get_datatype_odim(datatype)`  
maps the config file radar data type name into the corresponding odim data type name and Py-ART field name

#### Parameters

**datatype** [str] config file radar data type name

#### Returns

**metranet type** [dict] dictionary containing the odim data type name and its corresponding Py-ART field name

`pyrad.io.io_aux.get_datetime(fname, datadescriptor)`  
Given a data descriptor gets date and time from file name

#### Parameters

**fname** [str] file name

**datadescriptor** [str] radar field type. Format : [radar file type]:[datatype]

#### Returns

**fdatetime** [datetime object] date and time in file name

`pyrad.io.io_aux.get_field_name(datatype)`

Return long name of datatype.

**Parameters**

**datatype** [str] The data type

**Returns**

**name** [str] The name

`pyrad.io.io_aux.get_field_unit(datatype)`

Return unit of datatype.

**Parameters**

**datatype** [str] The data type

**Returns**

**unit** [str] The unit

`pyrad.io.io_aux.get_fieldname_cosmo(field_name)`

maps the Py-ART field name into the corresponding COSMO variable name

**Parameters**

**field\_name** [str] Py-ART field name

**Returns**

**cosmo\_name** [str] Py-ART variable name

`pyrad.io.io_aux.get_fieldname_pyart(datatype)`

maps the config file radar data type name into the corresponding rainbow Py-ART field name

**Parameters**

**datatype** [str] config file radar data type name

**Returns**

**field\_name** [str] Py-ART field name

`pyrad.io.io_aux.get_file_list(datadescriptor, starttimes, endtimes, cfg, scan=None)`

gets the list of files with a time period

**Parameters**

**datadescriptor** [str] radar field type. Format : [radar file type]:[datatype]

**starttimes** [array of datetime objects] start of time periods

**endtimes** [array of datetime object] end of time periods

**cfg: dictionary of dictionaries** configuration info to figure out where the data is

**scan** [str] scan name

**Returns**

**filelist** [list of strings] list of files within the time period

`pyrad.io.io_aux.get_new_rainbow_file_name(master_fname, master_datadescriptor, datatype)`

get the rainbow file name containing datatype from a master file name and data type

**Parameters**

**master\_fname** [str] the master file name

**master\_datadescriptor** [str] the master data type descriptor

**datatype** [str] the data type of the new file name to be created

#### Returns

**new\_fname** [str] the new file name

`pyrad.io.io_aux.get_rad4alp_dir(basepath, voltime, radar_name='A', radar_res='L', scan='001', path_convention='MCH')`  
gets the directory where rad4alp data is stored

#### Parameters

**basepath** [str] base path

**voltime** [datetime object] nominal time

**radar\_name** [str] radar name (A, D, L, P, W)

**radar\_res** [str] radar resolution (H, L)

**scan** [str] scan

**path\_convention** [str] The path convention. Can be 'LTE', 'MCH' or 'RT'

#### Returns

**datapath** [str] The data path

**basename** [str] The base name. ex: PHA17213

`pyrad.io.io_aux.get_rad4alp_grid_dir(basepath, voltime, datatype, acronym, path_convention='MCH')`  
gets the directory where rad4alp grid data is stored

#### Parameters

**basepath** [str] base path

**voltime** [datetime object] nominal time

**datatype** [str] data type

**acronym** [str] acronym identifying the data type

**path\_convention** [str] The path convention. Can be 'LTE', 'MCH' or 'RT'

#### Returns

**datapath** [str] The data path

`pyrad.io.io_aux.get_rad4alp_prod_fname(datatype)`  
Given a datatype find the corresponding start and termination of the METRANET product file name

#### Parameters

**datatype** [str] the data type

#### Returns

**acronym** [str] The start of the METRANET file name

**termination** [str] The end of the METRANET file name

`pyrad.io.io_aux.get_save_dir(basepath, procname, dsname, prdname, timeinfo=None, timeformat='%Y-%m-%d', create_dir=True)`  
obtains the path to a product directory and eventually creates it

#### Parameters

**basepath** [str] product base path

**procname** [str] name of processing space

**dsname** [str] data set name

**prdname** [str] product name

**timeinfo** [datetime] time info to generate the date directory. If None there is no time format in the path

**timeformat** [str] Optional. The time format.

**create\_dir** [boolean] If True creates the directory

#### Returns

**savedir** [str] path to product

`pyrad.io.io_aux.get_scan_list(scandescrptor_list)`  
determine which is the scan list for each radar

#### Parameters

**scandescrptor** [list of string] the list of all scans for all radars

#### Returns

**scan\_list** [list of lists] the list of scans corresponding to each radar

`pyrad.io.io_aux.get_trtfile_list(basepath, starttime, endtime)`  
gets the list of TRT files with a time period

#### Parameters

**datapath** [str] directory where to look for data

**starttime** [datetime object] start of time period

**endtime** [datetime object] end of time period

#### Returns

**filelist** [list of strings] list of files within the time period

`pyrad.io.io_aux.make_filename(prdtype, dstype, dsname, ext_list, prdcfginfo=None, timeinfo=None, timeformat='%Y%m%d%H%M%S', runinfo=None)`  
creates a product file name

#### Parameters

**timeinfo** [datetime] time info to generate the date directory

**prdtype** [str] product type, i.e. 'ppi', etc.

**dstype** [str] data set type, i.e. 'raw', etc.

**dsname** [str] data set name

**ext\_list** [list of str] file name extensions, i.e. 'png'

**prdcfginfo** [str] Optional. string to add product configuration information, i.e. 'el0.4'

**timeformat** [str] Optional. The time format

**runinfo** [str] Optional. Additional information about the test (e.g. 'RUN01', 'TS011')

#### Returns

**fname\_list** [list of str] list of file names (as many as extensions)

`pyrad.io.io_aux.map_Doppler(Doppler_data_bin, Nyquist_vel)`  
maps the binary METRANET Doppler data to actual Doppler velocity

**Parameters**

**Doppler\_data\_bin** [numpy array] The binary METRANET data

**Returns**

**Doppler\_data** [numpy array] The Doppler velocity in [m/s]

`pyrad.io.io_aux.map_hydro(hydro_data_op)`  
maps the operational hydrometeor classification identifiers to the ones used by Py-ART

**Parameters**

**hydro\_data\_op** [numpy array] The operational hydrometeor classification data

**Returns**

**hydro\_data\_py** [numpy array] The pyart hydrometeor classification data

## PYRAD.IO.CONFIG

Functions for reading pyrad config files

<i>read_config</i> (fname[, cfg])	Read a pyrad config file.
<i>get_num_elements</i> (dtype, nelstr)	Checks if data type is an array or a structure.
<i>string_to_datatype</i> (dtype, strval)	Converts a string containing a value into its Python value
<i>get_array</i> (cfgfile, pos, nel, valtype)	reads an array in a config file
<i>get_struct</i> (cfgfile, pos, nels, fname)	reads an struct in a config file
<i>get_array_type</i> (dtype)	Determines Python array type from the config file array type
<i>init_array</i> (nel, dtype)	Initializes a Python array

`pyrad.io.config.get_array` (*cfgfile*, *pos*, *nel*, *valtype*)  
reads an array in a config file

### Parameters

**cfgfile** [file object] config file  
**pos** [int] position in file object  
**nel** [int] number of elements of the ray  
**valtype** [str] type of array

### Returns

**arr** [array] array values  
**newpos** [int] new position in file object

`pyrad.io.config.get_array_type` (*dtype*)  
Determines Python array type from the config file array type

### Parameters

**dtype** [str] config file data type

### Returns

**pytype** [str] Python array type

`pyrad.io.config.get_num_elements` (*dtype*, *nelstr*)  
Checks if data type is an array or a structure.

### Parameters

**dtype** [str] data type specifier

**nelstr** [str] number of elements

#### Returns

**nel** [int] number of elements if type is \*ARR or STRUCT. 0 otherwise

**isstruct** [bool] true if the type is STRUCT

`pyrad.io.config.get_struct(cfgfile, pos, nels, fname)`  
reads an struct in a config file

#### Parameters

**cfgfile** [file object] config file

**pos** [int] position in file object

**nel** [int] number of elements of the ray

**fname** [str] config file name

#### Returns

**struct** [dict] dictionary of struct values

**newpos** [int] new position in file object

`pyrad.io.config.init_array(nel, dtype)`  
Initializes a Python array

#### Parameters

**nel** [int] number of elements in the array

**dtype** [str] config file data type

#### Returns

**pyarr** [array] Python array

`pyrad.io.config.read_config(fname, cfg=None)`  
Read a pyrad config file.

#### Parameters

**fname** [str] Name of the configuration file to read.

**cfg** [dict of dicts, optional] dictionary of dictionaries containing configuration parameters where the new parameters will be placed

#### Returns

**cfg** [dict of dicts] dictionary of dictionaries containing the configuration parameters

`pyrad.io.config.string_to_datatype(dtype, strval)`  
Converts a string containing a value into its Python value

#### Parameters

**dtype** [str] data type specifier

**strval** [str] string value

#### Returns

**val** [scalar] value contained in the string



## PYRAD.IO.READ\_DATA\_RADAR

Functions for reading radar data files

<i>get_data</i> (voltime, datatypesdescr, cfg)	Reads pyrad input data.
<i>merge_scans_rainbow</i> (basepath, scan_list, ...)	merge rainbow scans
<i>merge_scans_psr</i> (basepath, basepath_psr, ...)	merge rainbow scans
<i>merge_scans_psr_spectra</i> (basepath, ...[, radarnr])	merge rainbow scans
<i>merge_scans_dem</i> (basepath, scan_list, ...[, ...])	merge rainbow scans
<i>merge_scans_rad4alp</i> (basepath, scan_list, ...)	merge rad4alp data.
<i>merge_scans_odim</i> (basepath, scan_list, ...[, ...])	merge odim data.
<i>merge_scans_nexrad2</i> (basepath, scan_list, ...)	merge NEXRAD level 2 data.
<i>merge_scans_cfradial2</i> (basepath, scan_list, ...)	merge CFRADIAL2 data.
<i>merge_scans_cf1</i> (basepath, scan_list, ...[, ...])	merge CF1 data.
<i>merge_scans_cosmo</i> (voltime, datatype_list, cfg)	merge rainbow scans
<i>merge_scans_cosmo_rad4alp</i> (voltime, datatype, cfg)	merge cosmo rad4alp scans.
<i>merge_scans_dem_rad4alp</i> (voltime, datatype, cfg)	merge DEM rad4alp scans.
<i>merge_scans_other_rad4alp</i> (voltime, datatype, cfg)	merge other rad4alp polar products not contained in the basic M or P files, i.e.
<i>merge_scans_iq_rad4alp</i> (basepath, ...[, ...])	merge rad4alp IQ scans
<i>merge_fields_rainbow</i> (basepath, scan_name, ...)	merge Rainbow fields into a single radar object.
<i>merge_fields_psr</i> (basepath, basepath_psr, ...)	merge Rainbow fields into a single radar object.
<i>merge_fields_psr_spectra</i> (basepath, ...[, ...])	merge Rainbow fields into a single radar object.
<i>merge_fields_rad4alp_grid</i> (voltime, ...[, ...])	merge rad4alp Cartesian products
<i>merge_fields_pyrad</i> (basepath, loadname, ...)	merge fields from Pyrad-generated files into a single radar object.
<i>merge_fields_pyradcosmo</i> (basepath, voltime, ...)	merge fields from Pyrad-generated files into a single radar object.
<i>merge_fields_pyradgrid</i> (basepath, loadname, ...)	merge fields from Pyrad-generated files into a single radar object.
<i>merge_fields_pyrad_spectra</i> (basepath, ...[, ...])	merge fields from Pyrad-generated files into a single radar spectra object.
<i>merge_fields_dem</i> (basepath, scan_name, ...)	merge DEM fields into a single radar object.
<i>merge_fields_cosmo</i> (filename_list)	merge COSMO fields in Rainbow file format

Continued on next page

Table 1 – continued from previous page

<code>get_data_rainbow(filename, datatype)</code>	gets rainbow radar data
<code>get_data_rad4alp(filename, datatype_list, ...)</code>	gets rad4alp radar data
<code>get_data_odim(filename, datatype_list, ...)</code>	gets ODIM radar data
<code>add_field(radar_dest, radar_orig)</code>	adds the fields from orig radar into dest radar.
<code>interpol_field(radar_dest, radar_orig, ...)</code>	interpolates field field_name contained in radar_orig to the grid in radar_dest
<code>crop_grid(grid[, lat_min, lat_max, lon_min, ...])</code>	crops a grid object
<code>merge_grids(grid1, grid2)</code>	Merges two grids

`pyrad.io.read_data_radar.add_field(radar_dest, radar_orig)`

adds the fields from orig radar into dest radar. If they are not in the same grid, interpolates them to dest grid

#### Parameters

**radar\_dest** [radar object] the destination radar

**radar\_orig** [radar object] the radar object containing the original field

#### Returns

**field\_dest** [dict] interpolated field and metadata

`pyrad.io.read_data_radar.crop_grid(grid, lat_min=None, lat_max=None, lon_min=None, lon_max=None, alt_min=None, alt_max=None)`

crops a grid object

#### Parameters

**grid** [grid object] the grid object to crop

**lat\_min, lat\_max, lon\_min, lon\_max** [float] the lat/lon limits of the object (deg)

**alt\_min, alt\_max** [float] the altitude limits of the object (m MSL)

#### Returns

**grid\_crop** [grid object] The cropped grid

`pyrad.io.read_data_radar.get_data(voltime, datatypesdescr, cfg)`

Reads pyrad input data.

#### Parameters

**voltime** [datetime object] volume scan time

**datatypesdescr** [list] list of radar field types to read. Format :  
[radarnr]:[datagroup]:[datatype],[dataset],[product] 'dataset' is only specified for data groups 'ODIM', 'CFRADIAL', 'CFRADIAL2', 'CF1', 'ODIMPYRAD' 'PYRAD-GRID' and 'NETCDFSPECTRA'. 'product' is only specified for data groups 'CFRADIAL', 'ODIMPYRAD', 'PYRADGRID' and 'NETCDFSPECTRA' The data group specifies the type file from which data is extracted. It can be:

'RAINBOW': Proprietary Leonardo format 'COSMO': COSMO model data saved in Rainbow file format 'DEM': Visibility data saved in Rainbow file format 'PSR': Reads PSR data file to extract range gate information

(Noise and transmitted power)

'RAD4ALP': METRANET format used for the operational MeteoSwiss data. To find out which datatype to use to match a particular METRANET field name check the function 'get\_datatype\_metrnet' in `pyrad/io/io_aux.py`

**‘RAD4ALPCOSMO’:** COSMO model data saved in a binary file format.

Used by operational MeteoSwiss radars

**‘RAD4ALPDEM’:** Visibility data saved in a binary format used by operational MeteoSwiss radars

**‘RAD4ALPHYDRO’:** Used to read the MeteoSwiss operational hydrometeor classification

**‘RAD4ALPDOPPLER’:** Used to read the MeteoSwiss operational dealiased Doppler velocity

**‘ODIM’:** Generic ODIM file format. For such types ‘dataset’ specifies the directory and file name date convention. Example: ODIM:dBZ,D{%Y-%m-%d}-F{%Y%m%d%H%M%S}. To find out which datatype to use to match a particular ODIM field name check the function ‘get\_datatype\_odim’ in pyrad/io/io\_aux.py

**‘NEXRADII’:** Nexrad-level II file format.

**‘CFRADIAL2’:** CFRADIAL2 file format. For such types ‘dataset’ specifies the directory and file name date convention. Example: ODIM:dBZ,D{%Y-%m-%d}-F{%Y%m%d%H%M%S}. To find out which datatype to use to match a particular ODIM field name check the function ‘get\_datatype\_odim’ in pyrad/io/io\_aux.py

**‘CF1’:** CF1 file format. For such types ‘dataset’ specifies the directory and file name date convention. Example: ODIM:dBZ,D{%Y-%m-%d}-F{%Y%m%d%H%M%S}. To find out which datatype to use to match a particular ODIM field name check the function ‘get\_datatype\_odim’ in pyrad/io/io\_aux.py

**‘MXPOL’:** MXPOL (EPFL) data written in a netcdf file

**‘CFRADIAL’:** CFRadial format with the naming convention and directory structure in which Pyrad saves the data. For such datatypes ‘dataset’ specifies the directory where the dataset is stored and ‘product’ specifies the directory where the product is stored. Example: CFRADIAL:dBZc,Att\_ZPhi,SAVEVOL\_dBZc

**‘CFRADIALCOSMO’:** COSMO data in radar coordinates in a CFRadial file format. **‘ODIMPYRAD’:** ODIM file format with the naming convention and

directory structure in which Pyrad saves the data. For such datatypes ‘dataset’ specifies the directory where the dataset is stored and ‘product’ specifies the directory where the product is stored. Example: ODIMPYRAD:dBZc,Att\_ZPhi,SAVEVOL\_dBZc

**‘RAD4ALPGRID’:** METRANET format used for the operational MeteoSwiss Cartesian products.

**‘RAD4ALPGIF’:** Format used for operational MeteoSwiss Cartesian products stored as gif files

**‘PYRADGRID’:** Pyrad generated Cartesian grid products. For such datatypes ‘dataset’ specifies the directory where the dataset is stored and ‘product’ specifies the directory where the product is stored. Example: ODIMPYRAD:RR,RZC,SAVEVOL

**‘PSRSPECTRA’:** Format used to store Rainbow power spectra recordings.

**‘NETCDFSPECTRA’**: Format analogous to CFRadial and used to store  
Doppler spectral

**‘RAD4ALPIQ’**: Format used to store rad4alp IQ data

‘RAINBOW’, ‘RAD4ALP’, ‘ODIM’, ‘CFRADIAL2’, ‘CF1’ and ‘MXPOL’ are primary data file sources and they cannot be mixed for the same radar. It is also the case for their complementary data files, i.e. ‘COSMO’ and ‘RAD4ALPCOSMO’, etc. ‘CFRADIAL’ and ‘ODIMPYRAD’ are secondary data file sources and they can be combined with any other datagroup type. For a list of accepted datatypes and how they map to the Py-ART name convention check function ‘get\_field\_name\_pyart’ in pyrad/io/io\_aux.py

**cfg: dictionary of dictionaries** configuration info to figure out where the data is

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.get_data_mxpola(filename, datatype_list)`  
gets MXPOL radar data

#### Parameters

**filename** [str] name of file containing MXPOL data

**datatype\_list** [list of strings] list of data fields to get

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.get_data_odim(filename, datatype_list, scan_name, cfg, ind_rad=0)`  
gets ODIM radar data

#### Parameters

**filename** [str] name of file containing odim data

**datatype\_list** [list of strings] list of data fields to get

**scan\_name** [str] name of the elevation (001 to 020)

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object. None if the reading has not been successful

`pyrad.io.read_data_radar.get_data_rad4alpa(filename, datatype_list, scan_name, cfg, ind_rad=0)`  
gets rad4alp radar data

#### Parameters

**filename** [str] name of file containing rainbow data

**datatype\_list** [list of strings] list of data fields to get

**scan\_name** [str] name of the elevation (001 to 020)

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object. None if the reading has not been successful

`pyrad.io.read_data_radar.get_data_rainbow(filename, datatype)`  
gets rainbow radar data

#### Parameters

**filename** [str] name of file containing rainbow data

**datatype** [str] field name

#### Returns

**radar** [Radar or None] radar object if the reading of the data has been successful. None otherwise

`pyrad.io.read_data_radar.interpolate_field(radar_dest, radar_orig, field_name, fill_value=None, ang_tol=0.5)`  
interpolates field field\_name contained in radar\_orig to the grid in radar\_dest

#### Parameters

**radar\_dest** [radar object] the destination radar

**radar\_orig** [radar object] the radar object containing the original field

**field\_name: str** name of the field to interpolate

**fill\_value: float** The fill value

**ang\_tol** [float] angle tolerance to determine whether the radar origin sweep is the radar destination sweep

#### Returns

**field\_dest** [dict] interpolated field and metadata

`pyrad.io.read_data_radar.merge_fields_cosmo(filename_list)`  
merge COSMO fields in Rainbow file format

#### Parameters

**filename\_list** [str] list of file paths where to find the data

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_fields_dem(basepath, scan_name, datatype_list)`  
merge DEM fields into a single radar object.

#### Parameters

**basepath** [str] name of the base path where to find the data

**scan\_name: str** name of the scan

**datatype\_list** [list] lists of data types to get

#### Returns

**radar** [Radar] radar object

```
pyrad.io.read_data_radar.merge_fields_psr(basepath,      basepath_psr,      scan_name,
                                           voltime,    datatype_list,  undo_txcorr=True,
                                           cpi='low_prf', ang_tol=0.5,  azi_min=None,
                                           azi_max=None,          ele_min=None,
                                           ele_max=None,          rng_min=None,
                                           rng_max=None)
```

merge Rainbow fields into a single radar object.

#### Parameters

**basepath** [str] name of the base path where to find the data

**basepath\_psr** [str] name of the base path where to find the PSR data

**scan\_name: str** name of the scan

**voltime** [datetime object] reference time of the scan

**datatype\_list** [list] lists of data types to get

**undo\_txcorr** [Bool] If true the correction for transmitted power is undone when getting the noise

**cpi** [str] The CPI to use. Can be 'low\_prf', 'intermediate\_prf', 'high\_prf', 'mean', 'all'. If 'mean' the mean within the angle step is taken

**ang\_tol** [float] Tolerated angle distance between nominal radar angle and angle in PSR files

**azi\_min, azi\_max, ele\_min, ele\_max** [float or None] The minimum and maximum angles to keep (deg)

**rng\_min, rng\_max** [float or None] The minimum and maximum ranges to keep (m)

#### Returns

**radar** [Radar] radar object

```
pyrad.io.read_data_radar.merge_fields_psr_spectra(basepath,      basepath_psr,
                                                    scan_name, voltime, datatype_list,
                                                    undo_txcorr=True, fold=True, pos-
                                                    itive_away=True,  cpi='low_prf',
                                                    ang_tol=0.5,      azi_min=None,
                                                    azi_max=None,      ele_min=None,
                                                    ele_max=None,      rng_min=None,
                                                    rng_max=None)
```

merge Rainbow fields into a single radar object.

#### Parameters

**basepath** [str] name of the base path where to find the data

**basepath\_psr** [str] name of the base path where to find the PSR data

**scan\_name: str** name of the scan

**voltime** [datetime object] reference time of the scan

**datatype\_list** [list] lists of data types to get

**undo\_txcorr: Bool** If True the correction of the transmitted power is removed from the noise signal

**fold: Bool** If True the spectra is folded so that 0-Doppler is in the middle

**positive\_away: Bool** If True the spectra is reversed so that positive velocities are away from the radar

**cpi** [str] The CPI to use. Can be 'low\_prf', 'intermediate\_prf', 'high\_prf' or 'all'

**ang\_tol** [float] Tolerated angle distance between nominal radar angle and angle in PSR files

**azi\_min, azi\_max, ele\_min, ele\_max** [float or None] The minimum and maximum angles to keep (deg)

**rng\_min, rng\_max** [float or None] The minimum and maximum ranges to keep (m)

#### Returns

**psr** [radar spectra object] radar spectra object

`pyrad.io.read_data_radar.merge_fields_pyrad` (*basepath, loadname, voltime, datatype\_list, dataset\_list, product\_list, rng\_min=None, rng\_max=None, azi\_min=None, azi\_max=None, ele\_min=None, ele\_max=None, termination='.nc'*)

merge fields from Pyrad-generated files into a single radar object. Accepted file types are CFRadial and ODIM.

#### Parameters

**basepath** [str] name of the base path where to find the data

**loadname: str** name of the saving directory

**voltime** [datetime object] reference time of the scan

**datatype\_list** [list] list of data types to get

**dataset\_list** [list] list of datasets that produced the data type to get. Used to get path.

**product\_list** [list] list of products. Used to get path

**rng\_min, rng\_max** [float] The range limits [m]. If None the entire coverage of the radar is going to be used

**ele\_min, ele\_max, azi\_min, azi\_max** [float or None] The limits of the grid [deg]. If None the limits will be the limits of the radar volume

**termination** [str] file termination type. Can be '.nc' or '.h\*'

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_fields_pyrad_spectra` (*basepath, loadname, voltime, datatype\_list, dataset\_list, product\_list, rng\_min=None, rng\_max=None, azi\_min=None, azi\_max=None, ele\_min=None, ele\_max=None, termination='.nc'*)

merge fields from Pyrad-generated files into a single radar spectra object. Accepted file types are netcdf

#### Parameters

**basepath** [str] name of the base path where to find the data

**loadname: str** name of the saving directory

**voltime** [datetime object] reference time of the scan

**datatype\_list** [list] list of data types to get

**dataset\_list** [list] list of datasets that produced the data type to get. Used to get path.

**product\_list** [list] list of products. Used to get path

**rng\_min, rng\_max** [float] The range limits [m]. If None the entire coverage of the radar is going to be used

**ele\_min, ele\_max, azi\_min, azi\_max** [float or None] The limits of the grid [deg]. If None the limits will be the limits of the radar volume

**termination** [str] file termination type. Can be '.nc' or '.h\*'

#### Returns

**radar** [Radar] radar object

```
pyrad.io.read_data_radar.merge_fields_pyradcosmo(basepath, voltime, datatype_list,  
                                                dataset_list, cfg, rng_min=None,  
                                                rng_max=None, azi_min=None,  
                                                azi_max=None, ele_min=None,  
                                                ele_max=None, termination='.nc')
```

merge fields from Pyrad-generated files into a single radar object. Accepted file types are CFRadial and ODIM.

#### Parameters

**basepath** [str] name of the base path where to find the data

**voltime** [datetime object] reference time of the scan

**datatype\_list** [list] list of data types to get

**dataset\_list** [list] list of datasets that produced the data type to get. Used to get path.

**cfg** [dictionary of dictionaries] configuration info

**rng\_min, rng\_max** [float] The range limits [m]. If None the entire coverage of the radar is going to be used

**ele\_min, ele\_max, azi\_min, azi\_max** [float or None] The limits of the grid [deg]. If None the limits will be the limits of the radar volume

**termination** [str] file termination type. Can be '.nc' or '.h\*'

#### Returns

**radar** [Radar] radar object

```
pyrad.io.read_data_radar.merge_fields_pyradgrid(basepath, loadname, voltime,  
                                                datatype_list, dataset_list, prod-  
                                                uct_list, cfg, termination='.nc')
```

merge fields from Pyrad-generated files into a single radar object. Accepted file types are CFRadial and ODIM.

#### Parameters

**basepath** [str] name of the base path where to find the data

**loadname: str** name of the saving directory

**voltime** [datetime object] reference time of the scan

**datatype\_list** [list] list of data types to get

**dataset\_list** [list] list of datasets that produced the data type to get. Used to get path.

**product\_list** [list] list of products. Used to get path

**cfg** [dict] dictionary containing configuration parameters

**termination** [str] file termination type. Can be '.nc' or '.h\*'



**Returns**

**grid** [Grid] grid object

`pyrad.io.read_data_radar.merge_fields_rad4alp_grid`(*voltime*, *datatype\_list*, *cfg*,  
*ind\_rad=0*, *ftype='METRANET'*)  
merge rad4alp Cartesian products

**Parameters**

**voltime: datetime object** reference time of the scan

**datatype** [str] name of the data type to read

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

**ftype** [str] File type. Can be 'METRANET', 'gif' or 'bin'

**Returns**

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_fields_rainbow`(*basepath*, *scan\_name*, *voltime*,  
*datatype\_list*)  
merge Rainbow fields into a single radar object.

**Parameters**

**basepath** [str] name of the base path where to find the data

**scan\_name: str** name of the scan

**voltime** [datetime object] reference time of the scan

**datatype\_list** [list] lists of data types to get

**Returns**

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_grids`(*grid1*, *grid2*)  
Merges two grids

**Parameters**

**grid1, grid2** [grid object] the grid objects to merge

**Returns**

**grid** [grid object] The merged grid

`pyrad.io.read_data_radar.merge_scans_cf1`(*basepath*, *scan\_list*, *radar\_name*, *radar\_res*,  
*voltime*, *datatype\_list*, *dataset\_list*, *cfg*,  
*ind\_rad=0*)  
merge CF1 data.

**Parameters**

**basepath** [str] base path of CF1 radar data

**scan\_list** [list] list of scans

**voltime: datetime object** reference time of the scan

**datatype\_list** [list] lists of data types to get

**dataset\_list** [list] list of datasets. Used to get path

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_scans_cfradial2` (*basepath, scan\_list, radar\_name, radar\_res, voltime, datatype\_list, dataset\_list, cfg, ind\_rad=0*)

merge CFRADIAL2 data.

#### Parameters

**basepath** [str] base path of CFRADIAL2 radar data

**scan\_list** [list] list of scans

**voltime: datetime object** reference time of the scan

**datatype\_list** [list] lists of data types to get

**dataset\_list** [list] list of datasets. Used to get path

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_scans_cosmo` (*voltime, datatype\_list, cfg, ind\_rad=0*)  
merge rainbow scans

#### Parameters

**voltime: datetime object** reference time of the scan

**datatype\_list** [list] lists of data types to get

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_scans_cosmo_rad4alp` (*voltime, datatype, cfg, ind\_rad=0*)  
merge cosmo rad4alp scans. If data for all the scans cannot be retrieved returns None

#### Parameters

**voltime: datetime object** reference time of the scan

**datatype** [str] name of the data type to read

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

```
pyrad.io.read_data_radar.merge_scans_dem(basepath, scan_list, datatype_list,  
                                         rng_min=None, rng_max=None, azi_min=None,  
                                         azi_max=None, ele_min=None, ele_max=None)
```

merge rainbow scans

#### Parameters

**basepath** [str] base path of rad4alp radar data

**scan\_list** [list] list of scans

**datatype\_list** [list] lists of data types to get

**radarnr** [str] radar identifier number

**rng\_min, rng\_max** [float] The range limits [m]. If None the entire coverage of the radar is going to be used

**ele\_min, ele\_max, azi\_min, azi\_max** [float or None] The limits of the grid [deg]. If None the limits will be the limits of the radar volume

#### Returns

**radar** [Radar] radar object

```
pyrad.io.read_data_radar.merge_scans_dem_rad4alp(voltime, datatype, cfg, ind_rad=0)
```

merge DEM rad4alp scans. If data for all the scans cannot be retrieved returns None

#### Parameters

**voltime: datetime object** reference time of the scan

**datatype** [str] name of the data type to read

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

```
pyrad.io.read_data_radar.merge_scans_iq_rad4alp(basepath, basepath_iq, scan_list,  
                                                radar_name, radar_res, voltime,  
                                                datatype_list, cfg, ang_tol=0.1,  
                                                ang_step=0.01, ind_rad=0)
```

merge rad4alp IQ scans

#### Parameters

**basepath** [str] base path of rad4alp radar data

**basepath\_iq** [str] base path of rad4alp IQ data

**scan\_list** [list] list of scans (001 to 020)

**radar\_name** [str] radar\_name (A, D, L, ...)

**radar\_res** [str] radar resolution (H or L)

**voltime: datetime object** reference time of the scan

**datatype\_list** [list] lists of data types to get

**cfg** [dict] configuration dictionary

**ang\_tol** [float] Tolerance between nominal elevation and actual elevation

**ang\_step** [float] The elevation angular step used when checking valid ray files

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_scans_mxpola(basepath, scan_list, voltime, datatype_list, cfg)`

merge rad4alp data.

#### Parameters

**basepath** [str] base path of mxpol radar data

**scan\_list** [list] list of scans, in the case of mxpol, the elevation or azimuth denoted as 005 or 090 (for 5 or 90 degrees elevation) or 330 (for 330 degrees azimuth respectively)

**voltime: datetime object** reference time of the scan

**datatype\_list** [list] lists of data types to get

**cfg** [dict] configuration dictionary

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_scans_nexrad2(basepath, scan_list, radar_name, radar_res, voltime, datatype_list, dataset_list, cfg, ind_rad=0)`

merge NEXRAD level 2 data.

#### Parameters

**basepath** [str] base path of nexrad radar data

**scan\_list** [list] list of scans

**voltime: datetime object** reference time of the scan

**datatype\_list** [list] lists of data types to get

**dataset\_list** [list] list of datasets. Used to get path

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_scans_odim(basepath, scan_list, radar_name, radar_res, voltime, datatype_list, dataset_list, cfg, ind_rad=0)`

merge odim data.

#### Parameters

**basepath** [str] base path of odim radar data

**scan\_list** [list] list of scans

**voltime: datetime object** reference time of the scan

**datatype\_list** [list] lists of data types to get

**dataset\_list** [list] list of datasets. Used to get path

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_scans_other_rad4alp` (*voltime*, *datatype*, *cfg*,  
*ind\_rad=0*)

merge other rad4alp polar products not contained in the basic M or P files, i.e. hydro, dealiased velocity or precip. If data for all the scans cannot be retrieved returns None

#### Parameters

**voltime: datetime object** reference time of the scan

**datatype** [str] name of the data type to read

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_scans_psr` (*basepath*, *basepath\_psr*, *scan\_list*,  
*voltime*, *scan\_period*, *datatype\_list*, *cfg*,  
*radarnr='RADAR001'*)

merge rainbow scans

#### Parameters

**basepath** [str] base path of rainbow radar data

**basepath\_psr** [str] name of the base path where to find the PSR data

**scan\_list** [list] list of scans

**voltime: datetime object** reference time of the scan

**scan\_period** [float] time from reference time where to look for other scans data

**datatype\_list** [list] lists of data types to get

**cfg** [dict] configuration dictionary

**radarnr** [str] radar identifier number

#### Returns

**radar** [Radar] radar object

`pyrad.io.read_data_radar.merge_scans_psr_spectra` (*basepath*, *basepath\_psr*, *scan\_list*,  
*voltime*, *scan\_period*, *datatype\_list*,  
*cfg*, *radarnr='RADAR001'*)

merge rainbow scans

#### Parameters

**basepath** [str] base path of rad4alp radar data

**basepath\_psr** [str] name of the base path where to find the PSR data

**scan\_list** [list] list of scans

**voltime: datetime object** reference time of the scan

**scan\_period** [float] time from reference time where to look for other scans data

**datatype\_list** [list] lists of data types to get

**cfg** [dict] configuration dictionary

**radarnr** [str] radar identifier number

#### Returns

**radar** [Radar] radar object

```
pyrad.io.read_data_radar.merge_scans_rad4alp(basepath, scan_list, radar_name,  
                                              radar_res, voltime, datatype_list, cfg,  
                                              ind_rad=0)
```

merge rad4alp data.

#### Parameters

**basepath** [str] base path of rad4alp radar data

**scan\_list** [list] list of scans (001 to 020)

**radar\_name** [str] radar\_name (A, D, L, ...)

**radar\_res** [str] radar resolution (H or L)

**voltime: datetime object** reference time of the scan

**datatype\_list** [list] lists of data types to get

**cfg** [dict] configuration dictionary

**ind\_rad** [int] radar index

#### Returns

**radar** [Radar] radar object

```
pyrad.io.read_data_radar.merge_scans_rainbow(basepath, scan_list, voltime, scan_period,  
                                              datatype_list, cfg, radarnr='RADAR001')
```

merge rainbow scans

#### Parameters

**basepath** [str] base path of rad4alp radar data

**scan\_list** [list] list of scans

**voltime: datetime object** reference time of the scan

**scan\_period** [float] time from reference time where to look for other scans data

**datatype\_list** [list] lists of data types to get

**cfg** [dict] configuration dictionary

**radarnr** [str] radar identifier number

#### Returns

**radar** [Radar] radar object

## PYRAD.IO.READ\_DATA\_MXPOL

Functions for reading radar mxpol data files .. autosummary:

```
:toctree: generated/
classes - MXPOL:
    pyrad_MXPOL
classes - MCH:
    pyrad_MCH
utilities - read:
    row_stack
    findTimes
    int2float_radar
    readMXPOLRadData
    readCHRadData
utilities - config:
    load_myconfig
    get_mymetadata
    get_elevation_metadata
    generate_radar_table
    generate_polvar_metadata
    convert_polvar_name
```

`pyrad.io.read_data_mxpolar.convert_polvar_name` (*convention, polvar*)

Finds the correct variable name for a given convention (MXPOL, MCH) and a given variable name which was spelled with a different case or according to a different convention. For example, MXPOL convention uses 'Z' for the reflectivity variable, but if a user inserted 'Zh' this function will convert it to 'Z'. Parameters ——— convention : str, destination convention; either MCH or LTE polvar : str, key of polarimetric variable to be converted Returns ——— mykey : str, polarimetric variable key as used within the ProfileLab

toolbox context

`pyrad.io.read_data_mxpolar.findTimes` (*num\_sweep*)

Finds the times at the beginning and at the end of each sweep. Information comes from the elapsed time since the beginning of the volume scan, from the Rad4Alp: Specifications/ Request for Proposal (RFP) document.

Inputs ——— num\_sweep: int

rank of the sweep

**elapsed\_times[num\_sweep][0]: float** the elapsed time since the beginning of the volume scan at the beginning of the sweep

**elapsed\_times[num\_sweep][1]: float** the elapsed time since the beginning of the volume scan at the end of the sweep

`pyrad.io.read_data_mxpolar.generate_polvar_metadata` (*polvar, filename=None*)

Generates a dictionary with metadata for a polarimetric variable Parameters ——— polvar: str

polatimetric variable of interest

**filename: str** Filename of the configuration file. If None the default configuration file is loaded from the directory.

**polvar\_metadata: dict** dictionary with metatdata for polarimetric variable of interest

`pyrad.io.read_data_mxpul.generate_radar_table(radarname, filename=None)`

Generates a table with basic radar info, based on the given (or default) configfile Parameters ——— radarname: str

name of the radar (i.e. 'ALB' or 'A', 'MXPOL' etc)

**filename: str** path and name of the configfile, if None, the default configfile is used

**radar\_table: dict** table containing basic radar info

`pyrad.io.read_data_mxpul.get_elevation_metadata(radarname, filename=None)`

Gets the elevation angles for each sweep from the configuration file Inputs ——— radarname: str

name of the radar for which to retrieve elevation angles

**filename: str** name of the configuration file, if None, the default configuration file is used

**\_DEFAULT\_RADAR\_INFO['elevations'][radarname]: list** list of elevation angles in degrees

or None if not available

`pyrad.io.read_data_mxpul.get_mymetadata(p, filename=None)`

Return a dictionary of metadata for a given parameter, p. An empty dictionary will be returned if no metadata dictionary exists for parameter p. Parameters ——— p: str

parameter name (i.e. Polvar) for which to return metadata

**filename: str** Filename of the configuration file. If None the default configuration file is loaded from the directory.

**\_DEFAULT\_METADATA[p].copy(): dict** a copy of the parameter of interest from the metadata dictionary

`pyrad.io.read_data_mxpul.int2float_radar(data, varname, index_angle)`

Converts radar moments from bit to float Inputs ——— data: np.array

moment data as loaded from h5 file

**varname: str** name of the moment (i.e. 'ZH')

**index\_angle: int** rank of the sweep-1 (converted to base 0)

**output: np.array** moment data converted to float

`pyrad.io.read_data_mxpul.load_myconfig(filename=None)`

Load configuration from a config file. Parameters ——— filename: str

Filename of the configuration file. If None the default configuration file is loaded from the directory.

**\_DEFAULT\_METADATA: dict** Dictionary with metadata



```
class pyrad.io.read_data_mxppol.pyrad_IDL(filename, field_names=None, max_range=inf,  
                                           min_range=10000)  
    Bases: pyart.core.radar.Radar
```

## Methods

<i>add_field</i> (self, field_name, dic[, ...])	Add a field to the object.
<i>add_field_like</i> (self, existing_field_name, ...)	Add a field to the object with metadata from a existing field.
<i>check_field_exists</i> (self, field_name)	Check that a field exists in the fields dictionary.
<i>extract_sweeps</i> (self, sweeps)	Create a new radar contains only the data from select sweeps.
<i>get_azimuth</i> (self, sweep[, copy])	Return an array of azimuth angles for a given sweep.
<i>get_elevation</i> (self, sweep[, copy])	Return an array of elevation angles for a given sweep.
<i>get_end</i> (self, sweep)	Return the ending ray for a given sweep.
<i>get_field</i> (self, sweep, field_name[, copy])	Return the field data for a given sweep.
<i>get_gate_x_y_z</i> (self, sweep[, edges, ...])	Return the x, y and z gate locations in meters for a given sweep.
<i>get_nyquist_vel</i> (self, sweep[, check_uniform])	Return the Nyquist velocity in meters per second for a given sweep.
<i>get_slice</i> (self, sweep)	Return a slice for selecting rays for a given sweep.
<i>get_start</i> (self, sweep)	Return the starting ray index for a given sweep.
<i>get_start_end</i> (self, sweep)	Return the starting and ending ray for a given sweep.
<i>info</i> (self[, level, out])	Print information on radar.
<i>init_gate_altitude</i> (self)	Initialize the gate_altitude attribute.
<i>init_gate_longitude_latitude</i> (self)	Initialize or reset the gate_longitude and gate_latitude attributes.
<i>init_gate_x_y_z</i> (self)	Initialize or reset the gate_{x, y, z} attributes.
<i>init_rays_per_sweep</i> (self)	Initialize or reset the rays_per_sweep attribute.
<i>iter_azimuth</i> (self)	Return an iterator which returns sweep azimuth data.
<i>iter_elevation</i> (self)	Return an iterator which returns sweep elevation data.
<i>iter_end</i> (self)	Return an iterator over the sweep end indices.
<i>iter_field</i> (self, field_name)	Return an iterator which returns sweep field data.
<i>iter_slice</i> (self)	Return an iterator which returns sweep slice objects.
<i>iter_start</i> (self)	Return an iterator over the sweep start indices.
<i>iter_start_end</i> (self)	Return an iterator over the sweep start and end indices.

**\_\_class\_\_**

alias of builtins.type

**\_\_delattr\_\_** (self, name, /)

Implement delattr(self, name).

**\_\_dict\_\_** = mappingproxy({'\_\_module\_\_': 'pyrad.io.read\_data\_mxp<sub>pol</sub>', '\_\_init\_\_': <func

**\_\_dir\_\_** (self, /)

Default dir() implementation.

**\_\_eq\_\_** (self, value, /)

Return self==value.

**\_\_format\_\_** (self, format\_spec, /)

Default object formatter.

**\_\_ge\_\_** (*self*, *value*, /)

Return self>=value.

**\_\_getattr\_\_** (*self*, *name*, /)

Return getattr(self, name).

**\_\_getstate\_\_** (*self*)

Return object's state which can be pickled.

**\_\_gt\_\_** (*self*, *value*, /)

Return self>value.

**\_\_hash\_\_** (*self*, /)

Return hash(self).

**\_\_init\_\_** (*self*, *filename*, *field\_names=None*, *max\_range=inf*, *min\_range=10000*)

Initialize self. See help(type(self)) for accurate signature.

**\_\_init\_subclass\_\_** ()

This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

**\_\_le\_\_** (*self*, *value*, /)

Return self<=value.

**\_\_lt\_\_** (*self*, *value*, /)

Return self<value.

**\_\_module\_\_** = 'pyrad.io.read\_data\_mxp01'

**\_\_ne\_\_** (*self*, *value*, /)

Return self!=value.

**\_\_new\_\_** (\*args, \*\*kwargs)

Create and return a new object. See help(type) for accurate signature.

**\_\_reduce\_\_** (*self*, /)

Helper for pickle.

**\_\_reduce\_ex\_\_** (*self*, *protocol*, /)

Helper for pickle.

**\_\_repr\_\_** (*self*, /)

Return repr(self).

**\_\_setattr\_\_** (*self*, *name*, *value*, /)

Implement setattr(self, name, value).

**\_\_setstate\_\_** (*self*, *state*)

Restore unpicklable entries from pickled object.

**\_\_sizeof\_\_** (*self*, /)

Size of object in memory, in bytes.

**\_\_str\_\_** (*self*, /)

Return str(self).

**\_\_subclasshook\_\_** ()

Abstract classes can override this to customize issubclass().

This is invoked early on by `abc.ABCMeta.__subclasscheck__()`. It should return `True`, `False` or `NotImplemented`. If it returns `NotImplemented`, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

#### **`__weakref__`**

list of weak references to the object (if defined)

#### **`__check_sweep_in_range`** (*self*, *sweep*)

Check that a sweep number is in range.

#### **`__dic_info`** (*self*, *attr*, *level*, *out*, *dic=None*, *ident\_level=0*)

Print information on a dictionary attribute.

#### **`add_field`** (*self*, *field\_name*, *dic*, *replace\_existing=False*)

Add a field to the object.

#### **Parameters**

**`field_name`** [str] Name of the field to add to the dictionary of fields.

**`dic`** [dict] Dictionary contain field data and metadata.

**`replace_existing`** [bool] True to replace the existing field with key `field_name` if it exists, loosing any existing data. False will raise a `ValueError` when the field already exists.

#### **`add_field_like`** (*self*, *existing\_field\_name*, *field\_name*, *data*, *replace\_existing=False*)

Add a field to the object with metadata from a existing field.

Note that the data parameter is not copied by this method. If data refers to a 'data' array from an existing field dictionary, a copy should be made within or prior to using this method. If this is not done the 'data' key in both field dictionaries will point to the same NumPy array and modification of one will change the second. To copy NumPy arrays use the `copy()` method. See the Examples section for how to create a copy of the 'reflectivity' field as a field named 'reflectivity\_copy'.

#### **Parameters**

**`existing_field_name`** [str] Name of an existing field to take metadata from when adding the new field to the object.

**`field_name`** [str] Name of the field to add to the dictionary of fields.

**`data`** [array] Field data. A copy of this data is not made, see the note above.

**`replace_existing`** [bool] True to replace the existing field with key `field_name` if it exists, loosing any existing data. False will raise a `ValueError` when the field already exists.

## **Examples**

```
>>> radar.add_field_like('reflectivity', 'reflectivity_copy',
...                      radar.fields['reflectivity']['data'].copy())
```

#### **`check_field_exists`** (*self*, *field\_name*)

Check that a field exists in the fields dictionary.

If the field does not exist raise a `KeyError`.

#### **Parameters**

**`field_name`** [str] Name of field to check.

#### **`extract_sweeps`** (*self*, *sweeps*)

Create a new radar contains only the data from select sweeps.

**Parameters**

**sweeps** [array\_like] Sweeps (0-based) to include in new Radar object.

**Returns**

**radar** [Radar] Radar object which contains a copy of data from the selected sweeps.

**get\_azimuth** (*self, sweep, copy=False*)

Return an array of azimuth angles for a given sweep.

**Parameters**

**sweep** [int] Sweep number to retrieve data for, 0 based.

**copy** [bool, optional] True to return a copy of the azimuths. False, the default, returns a view of the azimuths (when possible), changing this data will change the data in the underlying Radar object.

**Returns**

**azimuths** [array] Array containing the azimuth angles for a given sweep.

**get\_elevation** (*self, sweep, copy=False*)

Return an array of elevation angles for a given sweep.

**Parameters**

**sweep** [int] Sweep number to retrieve data for, 0 based.

**copy** [bool, optional] True to return a copy of the elevations. False, the default, returns a view of the elevations (when possible), changing this data will change the data in the underlying Radar object.

**Returns**

**azimuths** [array] Array containing the elevation angles for a given sweep.

**get\_end** (*self, sweep*)

Return the ending ray for a given sweep.

**get\_field** (*self, sweep, field\_name, copy=False*)

Return the field data for a given sweep.

When used with *get\_gate\_x\_y\_z()* this method can be used to obtain the data needed for plotting a radar field with the correct spatial context.

**Parameters**

**sweep** [int] Sweep number to retrieve data for, 0 based.

**field\_name** [str] Name of the field from which data should be retrieved.

**copy** [bool, optional] True to return a copy of the data. False, the default, returns a view of the data (when possible), changing this data will change the data in the underlying Radar object.

**Returns**

**data** [array] Array containing data for the requested sweep and field.

**get\_gate\_x\_y\_z** (*self, sweep, edges=False, filter\_transitions=False*)

Return the x, y and z gate locations in meters for a given sweep.

With the default parameter this method returns the same data as contained in the *gate\_x*, *gate\_y* and *gate\_z* attributes but this method performs the gate location calculations only for the specified sweep and therefore is more efficient than accessing this data through these attribute.

When used with `get_field()` this method can be used to obtain the data needed for plotting a radar field with the correct spatial context.

#### Parameters

**sweep** [int] Sweep number to retrieve gate locations from, 0 based.

**edges** [bool, optional] True to return the locations of the gate edges calculated by interpolating between the range, azimuths and elevations. False (the default) will return the locations of the gate centers with no interpolation.

**filter\_transitions** [bool, optional] True to remove rays where the antenna was in transition between sweeps. False will include these rays. No rays will be removed if the `antenna_transition` attribute is not available (set to None).

#### Returns

**x, y, z** [2D array] Array containing the x, y and z, distances from the radar in meters for the center (or edges) for all gates in the sweep.

**get\_nyquist\_vel** (*self*, *sweep*, *check\_uniform=True*)

Return the Nyquist velocity in meters per second for a given sweep.

Raises a `LookupError` if the Nyquist velocity is not available, an `Exception` is raised if the velocities are not uniform in the sweep unless `check_uniform` is set to False.

#### Parameters

**sweep** [int] Sweep number to retrieve data for, 0 based.

**check\_uniform** [bool] True to check to perform a check on the Nyquist velocities that they are uniform in the sweep, False will skip this check and return the velocity of the first ray in the sweep.

#### Returns

**nyquist\_velocity** [float] Array containing the Nyquist velocity in m/s for a given sweep.

**get\_slice** (*self*, *sweep*)

Return a slice for selecting rays for a given sweep.

**get\_start** (*self*, *sweep*)

Return the starting ray index for a given sweep.

**get\_start\_end** (*self*, *sweep*)

Return the starting and ending ray for a given sweep.

**info** (*self*, *level='standard'*, *out=<\_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Print information on radar.

#### Parameters

**level** [{ 'compact', 'standard', 'full', 'c', 's', 'f' }] Level of information on radar object to print, compact is minimal information, standard more and full everything.

**out** [file-like] Stream to direct output to, default is to print information to standard out (the screen).

**init\_gate\_altitude** (*self*)

Initialize the `gate_altitude` attribute.

**init\_gate\_longitude\_latitude** (*self*)

Initialize or reset the `gate_longitude` and `gate_latitude` attributes.

**init\_gate\_x\_y\_z** (*self*)  
Initialize or reset the gate\_{x, y, z} attributes.

**init\_rays\_per\_sweep** (*self*)  
Initialize or reset the rays\_per\_sweep attribute.

**iter\_azimuth** (*self*)  
Return an iterator which returns sweep azimuth data.

**iter\_elevation** (*self*)  
Return an iterator which returns sweep elevation data.

**iter\_end** (*self*)  
Return an iterator over the sweep end indices.

**iter\_field** (*self*, *field\_name*)  
Return an iterator which returns sweep field data.

**iter\_slice** (*self*)  
Return an iterator which returns sweep slice objects.

**iter\_start** (*self*)  
Return an iterator over the sweep start indices.

**iter\_start\_end** (*self*)  
Return an iterator over the sweep start and end indices.

**class** pyrad.io.read\_data\_mxpole.pyrad\_MCH (*filename*, *field\_names=None*, *max\_range=inf*)  
Bases: pyart.core.radar.Radar

## Methods

<i>add_field</i> ( <i>self</i> , <i>field_name</i> , <i>dic</i> , ...) )	Add a field to the object.
<i>add_field_like</i> ( <i>self</i> , <i>existing_field_name</i> , ...)	Add a field to the object with metadata from a existing field.
<i>check_field_exists</i> ( <i>self</i> , <i>field_name</i> )	Check that a field exists in the fields dictionary.
<i>extract_sweeps</i> ( <i>self</i> , <i>sweeps</i> )	Create a new radar contains only the data from select sweeps.
<i>get_azimuth</i> ( <i>self</i> , <i>sweep</i> [, <i>copy</i> ])	Return an array of azimuth angles for a given sweep.
<i>get_elevation</i> ( <i>self</i> , <i>sweep</i> [, <i>copy</i> ])	Return an array of elevation angles for a given sweep.
<i>get_end</i> ( <i>self</i> , <i>sweep</i> )	Return the ending ray for a given sweep.
<i>get_field</i> ( <i>self</i> , <i>sweep</i> , <i>field_name</i> [, <i>copy</i> ])	Return the field data for a given sweep.
<i>get_gate_x_y_z</i> ( <i>self</i> , <i>sweep</i> [, <i>edges</i> , ...])	Return the x, y and z gate locations in meters for a given sweep.
<i>get_nyquist_vel</i> ( <i>self</i> , <i>sweep</i> [, <i>check_uniform</i> ])	Return the Nyquist velocity in meters per second for a given sweep.
<i>get_slice</i> ( <i>self</i> , <i>sweep</i> )	Return a slice for selecting rays for a given sweep.
<i>get_start</i> ( <i>self</i> , <i>sweep</i> )	Return the starting ray index for a given sweep.
<i>get_start_end</i> ( <i>self</i> , <i>sweep</i> )	Return the starting and ending ray for a given sweep.
<i>info</i> ( <i>self</i> [, <i>level</i> , <i>out</i> ])	Print information on radar.
<i>init_gate_altitude</i> ( <i>self</i> )	Initialize the gate_altitude attribute.
<i>init_gate_longitude_latitude</i> ( <i>self</i> )	Initialize or reset the gate_longitude and gate_latitude attributes.
<i>init_gate_x_y_z</i> ( <i>self</i> )	Initialize or reset the gate_{x, y, z} attributes.
<i>init_rays_per_sweep</i> ( <i>self</i> )	Initialize or reset the rays_per_sweep attribute.

Continued on next page

Table 2 – continued from previous page

<i>iter_azimuth</i> (self)	Return an iterator which returns sweep azimuth data.
<i>iter_elevation</i> (self)	Return an iterator which returns sweep elevation data.
<i>iter_end</i> (self)	Return an iterator over the sweep end indices.
<i>iter_field</i> (self, field_name)	Return an iterator which returns sweep field data.
<i>iter_slice</i> (self)	Return an iterator which returns sweep slice objects.
<i>iter_start</i> (self)	Return an iterator over the sweep start indices.
<i>iter_start_end</i> (self)	Return an iterator over the sweep start and end indices.

```

__class__
    alias of builtins.type

__delattr__(self, name, /)
    Implement delattr(self, name).

__dict__ = mappingproxy({'__module__': 'pyrad.io.read_data_mxp1', '__init__': <func
__dir__(self, /)
    Default dir() implementation.

__eq__(self, value, /)
    Return self==value.

__format__(self, format_spec, /)
    Default object formatter.

__ge__(self, value, /)
    Return self>=value.

__getattr__(self, name, /)
    Return getattr(self, name).

__getstate__(self)
    Return object's state which can be pickled.

__gt__(self, value, /)
    Return self>value.

__hash__(self, /)
    Return hash(self).

__init__(self, filename, field_names=None, max_range=inf)
    Initialize self. See help(type(self)) for accurate signature.

__init_subclass__()
    This method is called when a class is subclassed.

    The default implementation does nothing. It may be overridden to extend subclasses.

__le__(self, value, /)
    Return self<=value.

__lt__(self, value, /)
    Return self<value.

__module__ = 'pyrad.io.read_data_mxp1'

__ne__(self, value, /)
    Return self!=value.

```

**\_\_new\_\_** (\*args, \*\*kwargs)

Create and return a new object. See help(type) for accurate signature.

**\_\_reduce\_\_** (self, /)

Helper for pickle.

**\_\_reduce\_ex\_\_** (self, protocol, /)

Helper for pickle.

**\_\_repr\_\_** (self, /)

Return repr(self).

**\_\_setattr\_\_** (self, name, value, /)

Implement setattr(self, name, value).

**\_\_setstate\_\_** (self, state)

Restore unpicklable entries from pickled object.

**\_\_sizeof\_\_** (self, /)

Size of object in memory, in bytes.

**\_\_str\_\_** (self, /)

Return str(self).

**\_\_subclasshook\_\_** ()

Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.\_\_subclasscheck\_\_(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**\_\_check\_sweep\_in\_range** (self, sweep)

Check that a sweep number is in range.

**\_\_dic\_info** (self, attr, level, out, dic=None, ident\_level=0)

Print information on a dictionary attribute.

**add\_field** (self, field\_name, dic, replace\_existing=False)

Add a field to the object.

#### Parameters

**field\_name** [str] Name of the field to add to the dictionary of fields.

**dic** [dict] Dictionary contain field data and metadata.

**replace\_existing** [bool] True to replace the existing field with key field\_name if it exists, losing any existing data. False will raise a ValueError when the field already exists.

**add\_field\_like** (self, existing\_field\_name, field\_name, data, replace\_existing=False)

Add a field to the object with metadata from an existing field.

Note that the data parameter is not copied by this method. If data refers to a 'data' array from an existing field dictionary, a copy should be made within or prior to using this method. If this is not done the 'data' key in both field dictionaries will point to the same NumPy array and modification of one will change the second. To copy NumPy arrays use the copy() method. See the Examples section for how to create a copy of the 'reflectivity' field as a field named 'reflectivity\_copy'.

#### Parameters



**existing\_field\_name** [str] Name of an existing field to take metadata from when adding the new field to the object.

**field\_name** [str] Name of the field to add to the dictionary of fields.

**data** [array] Field data. A copy of this data is not made, see the note above.

**replace\_existing** [bool] True to replace the existing field with key field\_name if it exists, loosing any existing data. False will raise a ValueError when the field already exists.

## Examples

```
>>> radar.add_field_like('reflectivity', 'reflectivity_copy',  
...                      radar.fields['reflectivity']['data'].copy())
```

**check\_field\_exists** (*self*, *field\_name*)

Check that a field exists in the fields dictionary.

If the field does not exist raise a KeyError.

### Parameters

**field\_name** [str] Name of field to check.

**extract\_sweeps** (*self*, *sweeps*)

Create a new radar contains only the data from select sweeps.

### Parameters

**sweeps** [array\_like] Sweeps (0-based) to include in new Radar object.

### Returns

**radar** [Radar] Radar object which contains a copy of data from the selected sweeps.

**get\_azimuth** (*self*, *sweep*, *copy=False*)

Return an array of azimuth angles for a given sweep.

### Parameters

**sweep** [int] Sweep number to retrieve data for, 0 based.

**copy** [bool, optional] True to return a copy of the azimuths. False, the default, returns a view of the azimuths (when possible), changing this data will change the data in the underlying Radar object.

### Returns

**azimuths** [array] Array containing the azimuth angles for a given sweep.

**get\_elevation** (*self*, *sweep*, *copy=False*)

Return an array of elevation angles for a given sweep.

### Parameters

**sweep** [int] Sweep number to retrieve data for, 0 based.

**copy** [bool, optional] True to return a copy of the elevations. False, the default, returns a view of the elevations (when possible), changing this data will change the data in the underlying Radar object.

### Returns

**azimuths** [array] Array containing the elevation angles for a given sweep.

**get\_end** (*self*, *sweep*)

Return the ending ray for a given sweep.

**get\_field** (*self*, *sweep*, *field\_name*, *copy=False*)

Return the field data for a given sweep.

When used with *get\_gate\_x\_y\_z* () this method can be used to obtain the data needed for plotting a radar field with the correct spatial context.

#### Parameters

**sweep** [int] Sweep number to retrieve data for, 0 based.

**field\_name** [str] Name of the field from which data should be retrieved.

**copy** [bool, optional] True to return a copy of the data. False, the default, returns a view of the data (when possible), changing this data will change the data in the underlying Radar object.

#### Returns

**data** [array] Array containing data for the requested sweep and field.

**get\_gate\_x\_y\_z** (*self*, *sweep*, *edges=False*, *filter\_transitions=False*)

Return the x, y and z gate locations in meters for a given sweep.

With the default parameter this method returns the same data as contained in the *gate\_x*, *gate\_y* and *gate\_z* attributes but this method performs the gate location calculations only for the specified sweep and therefore is more efficient than accessing this data through these attribute.

When used with *get\_field* () this method can be used to obtain the data needed for plotting a radar field with the correct spatial context.

#### Parameters

**sweep** [int] Sweep number to retrieve gate locations from, 0 based.

**edges** [bool, optional] True to return the locations of the gate edges calculated by interpolating between the range, azimuths and elevations. False (the default) will return the locations of the gate centers with no interpolation.

**filter\_transitions** [bool, optional] True to remove rays where the antenna was in transition between sweeps. False will include these rays. No rays will be removed if the *antenna\_transition* attribute is not available (set to None).

#### Returns

**x, y, z** [2D array] Array containing the x, y and z, distances from the radar in meters for the center (or edges) for all gates in the sweep.

**get\_nyquist\_vel** (*self*, *sweep*, *check\_uniform=True*)

Return the Nyquist velocity in meters per second for a given sweep.

Raises a *LookupError* if the Nyquist velocity is not available, an *Exception* is raised if the velocities are not uniform in the sweep unless *check\_uniform* is set to False.

#### Parameters

**sweep** [int] Sweep number to retrieve data for, 0 based.

**check\_uniform** [bool] True to check to perform a check on the Nyquist velocities that they are uniform in the sweep, False will skip this check and return the velocity of the first ray in the sweep.

#### Returns

**nyquist\_velocity** [float] Array containing the Nyquist velocity in m/s for a given sweep.

**get\_slice** (*self*, *sweep*)

Return a slice for selecting rays for a given sweep.

**get\_start** (*self*, *sweep*)

Return the starting ray index for a given sweep.

**get\_start\_end** (*self*, *sweep*)

Return the starting and ending ray for a given sweep.

**info** (*self*, *level*=*'standard'*, *out*=<\_io.TextIOWrapper name=*'<stdout>' mode='w' encoding='UTF-8'>*)

Print information on radar.

#### Parameters

**level** [{*'compact'*, *'standard'*, *'full'*, *'c'*, *'s'*, *'f'*}] Level of information on radar object to print, compact is minimal information, standard more and full everything.

**out** [file-like] Stream to direct output to, default is to print information to standard out (the screen).

**init\_gate\_altitude** (*self*)

Initialize the gate\_altitude attribute.

**init\_gate\_longitude\_latitude** (*self*)

Initialize or reset the gate\_longitude and gate\_latitude attributes.

**init\_gate\_x\_y\_z** (*self*)

Initialize or reset the gate\_{x, y, z} attributes.

**init\_rays\_per\_sweep** (*self*)

Initialize or reset the rays\_per\_sweep attribute.

**iter\_azimuth** (*self*)

Return an iterator which returns sweep azimuth data.

**iter\_elevation** (*self*)

Return an iterator which returns sweep elevation data.

**iter\_end** (*self*)

Return an iterator over the sweep end indices.

**iter\_field** (*self*, *field\_name*)

Return an iterator which returns sweep field data.

**iter\_slice** (*self*)

Return an iterator which returns sweep slice objects.

**iter\_start** (*self*)

Return an iterator over the sweep start indices.

**iter\_start\_end** (*self*)

Return an iterator over the sweep start and end indices.

**class** pyrad.io.read\_data\_mxpole.pyrad\_MXPOL (*filename*, *field\_names*=None, *max\_range*=inf, *min\_range*=10000, *pyrad\_names*=True)

Bases: pyart.core.radar.Radar

#### Methods

<code>add_field(self, field_name, dic[, ...])</code>	Add a field to the object.
<code>add_field_like(self, existing_field_name, ...)</code>	Add a field to the object with metadata from a existing field.
<code>check_field_exists(self, field_name)</code>	Check that a field exists in the fields dictionary.
<code>extract_sweeps(self, sweeps)</code>	Create a new radar contains only the data from select sweeps.
<code>get_azimuth(self, sweep[, copy])</code>	Return an array of azimuth angles for a given sweep.
<code>get_elevation(self, sweep[, copy])</code>	Return an array of elevation angles for a given sweep.
<code>get_end(self, sweep)</code>	Return the ending ray for a given sweep.
<code>get_field(self, sweep, field_name[, copy])</code>	Return the field data for a given sweep.
<code>get_gate_x_y_z(self, sweep[, edges, ...])</code>	Return the x, y and z gate locations in meters for a given sweep.
<code>get_nyquist_vel(self, sweep[, check_uniform])</code>	Return the Nyquist velocity in meters per second for a given sweep.
<code>get_slice(self, sweep)</code>	Return a slice for selecting rays for a given sweep.
<code>get_start(self, sweep)</code>	Return the starting ray index for a given sweep.
<code>get_start_end(self, sweep)</code>	Return the starting and ending ray for a given sweep.
<code>info(self[, level, out])</code>	Print information on radar.
<code>init_gate_altitude(self)</code>	Initialize the gate_altitude attribute.
<code>init_gate_longitude_latitude(self)</code>	Initialize or reset the gate_longitude and gate_latitude attributes.
<code>init_gate_x_y_z(self)</code>	Initialize or reset the gate_{x, y, z} attributes.
<code>init_rays_per_sweep(self)</code>	Initialize or reset the rays_per_sweep attribute.
<code>iter_azimuth(self)</code>	Return an iterator which returns sweep azimuth data.
<code>iter_elevation(self)</code>	Return an iterator which returns sweep elevation data.
<code>iter_end(self)</code>	Return an iterator over the sweep end indices.
<code>iter_field(self, field_name)</code>	Return an iterator which returns sweep field data.
<code>iter_slice(self)</code>	Return an iterator which returns sweep slice objects.
<code>iter_start(self)</code>	Return an iterator over the sweep start indices.
<code>iter_start_end(self)</code>	Return an iterator over the sweep start and end indices.

**`__class__`**

alias of `builtins.type`

**`__delattr__`** (*self, name, /*)

Implement `delattr(self, name)`.

**`__dict__`** = `mappingproxy({'__module__': 'pyrad.io.read_data_mxpole', '__init__': <func`

**`__dir__`** (*self, /*)

Default `dir()` implementation.

**`__eq__`** (*self, value, /*)

Return `self==value`.

**`__format__`** (*self, format\_spec, /*)

Default object formatter.

**`__ge__`** (*self, value, /*)

Return `self>=value`.

**`__getattr__`** (*self, name, /*)

Return `getattr(self, name)`.

**\_\_getstate\_\_** (*self*)  
Return object's state which can be pickled.

**\_\_gt\_\_** (*self, value, /*)  
Return self>value.

**\_\_hash\_\_** (*self, /*)  
Return hash(self).

**\_\_init\_\_** (*self, filename, field\_names=None, max\_range=inf, min\_range=10000, pyrad\_names=True*)  
Initialize self. See help(type(self)) for accurate signature.

**\_\_init\_subclass\_\_** ()  
This method is called when a class is subclassed.  
  
The default implementation does nothing. It may be overridden to extend subclasses.

**\_\_le\_\_** (*self, value, /*)  
Return self<=value.

**\_\_lt\_\_** (*self, value, /*)  
Return self<value.

**\_\_module\_\_** = 'pyrad.io.read\_data\_mxp01'

**\_\_ne\_\_** (*self, value, /*)  
Return self!=value.

**\_\_new\_\_** (\*args, \*\*kwargs)  
Create and return a new object. See help(type) for accurate signature.

**\_\_reduce\_\_** (*self, /*)  
Helper for pickle.

**\_\_reduce\_ex\_\_** (*self, protocol, /*)  
Helper for pickle.

**\_\_repr\_\_** (*self, /*)  
Return repr(self).

**\_\_setattr\_\_** (*self, name, value, /*)  
Implement setattr(self, name, value).

**\_\_setstate\_\_** (*self, state*)  
Restore unpicklable entries from pickled object.

**\_\_sizeof\_\_** (*self, /*)  
Size of object in memory, in bytes.

**\_\_str\_\_** (*self, /*)  
Return str(self).

**\_\_subclasshook\_\_** ()  
Abstract classes can override this to customize issubclass().  
  
This is invoked early on by abc.ABCMeta.\_\_subclasscheck\_\_(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

**\_\_weakref\_\_**  
list of weak references to the object (if defined)

**\_\_check\_sweep\_in\_range** (*self, sweep*)  
Check that a sweep number is in range.

**\_dic\_info** (*self, attr, level, out, dic=None, ident\_level=0*)  
Print information on a dictionary attribute.

**add\_field** (*self, field\_name, dic, replace\_existing=False*)  
Add a field to the object.

#### Parameters

**field\_name** [str] Name of the field to add to the dictionary of fields.

**dic** [dict] Dictionary contain field data and metadata.

**replace\_existing** [bool] True to replace the existing field with key field\_name if it exists, loosing any existing data. False will raise a ValueError when the field already exists.

**add\_field\_like** (*self, existing\_field\_name, field\_name, data, replace\_existing=False*)  
Add a field to the object with metadata from a existing field.

Note that the data parameter is not copied by this method. If data refers to a 'data' array from an existing field dictionary, a copy should be made within or prior to using this method. If this is not done the 'data' key in both field dictionaries will point to the same NumPy array and modification of one will change the second. To copy NumPy arrays use the copy() method. See the Examples section for how to create a copy of the 'reflectivity' field as a field named 'reflectivity\_copy'.

#### Parameters

**existing\_field\_name** [str] Name of an existing field to take metadata from when adding the new field to the object.

**field\_name** [str] Name of the field to add to the dictionary of fields.

**data** [array] Field data. A copy of this data is not made, see the note above.

**replace\_existing** [bool] True to replace the existing field with key field\_name if it exists, loosing any existing data. False will raise a ValueError when the field already exists.

### Examples

```
>>> radar.add_field_like('reflectivity', 'reflectivity_copy',  
...                      radar.fields['reflectivity']['data'].copy())
```

**check\_field\_exists** (*self, field\_name*)  
Check that a field exists in the fields dictionary.

If the field does not exist raise a KeyError.

#### Parameters

**field\_name** [str] Name of field to check.

**extract\_sweeps** (*self, sweeps*)  
Create a new radar contains only the data from select sweeps.

#### Parameters

**sweeps** [array\_like] Sweeps (0-based) to include in new Radar object.

#### Returns

**radar** [Radar] Radar object which contains a copy of data from the selected sweeps.

**get\_azimuth** (*self, sweep, copy=False*)  
Return an array of azimuth angles for a given sweep.

**Parameters**

**sweep** [int] Sweep number to retrieve data for, 0 based.

**copy** [bool, optional] True to return a copy of the azimuths. False, the default, returns a view of the azimuths (when possible), changing this data will change the data in the underlying Radar object.

**Returns**

**azimuths** [array] Array containing the azimuth angles for a given sweep.

**get\_elevation** (*self*, *sweep*, *copy=False*)

Return an array of elevation angles for a given sweep.

**Parameters**

**sweep** [int] Sweep number to retrieve data for, 0 based.

**copy** [bool, optional] True to return a copy of the elevations. False, the default, returns a view of the elevations (when possible), changing this data will change the data in the underlying Radar object.

**Returns**

**azimuths** [array] Array containing the elevation angles for a given sweep.

**get\_end** (*self*, *sweep*)

Return the ending ray for a given sweep.

**get\_field** (*self*, *sweep*, *field\_name*, *copy=False*)

Return the field data for a given sweep.

When used with *get\_gate\_x\_y\_z()* this method can be used to obtain the data needed for plotting a radar field with the correct spatial context.

**Parameters**

**sweep** [int] Sweep number to retrieve data for, 0 based.

**field\_name** [str] Name of the field from which data should be retrieved.

**copy** [bool, optional] True to return a copy of the data. False, the default, returns a view of the data (when possible), changing this data will change the data in the underlying Radar object.

**Returns**

**data** [array] Array containing data for the requested sweep and field.

**get\_gate\_x\_y\_z** (*self*, *sweep*, *edges=False*, *filter\_transitions=False*)

Return the x, y and z gate locations in meters for a given sweep.

With the default parameter this method returns the same data as contained in the *gate\_x*, *gate\_y* and *gate\_z* attributes but this method performs the gate location calculations only for the specified sweep and therefore is more efficient than accessing this data through these attribute.

When used with *get\_field()* this method can be used to obtain the data needed for plotting a radar field with the correct spatial context.

**Parameters**

**sweep** [int] Sweep number to retrieve gate locations from, 0 based.

**edges** [bool, optional] True to return the locations of the gate edges calculated by interpolating between the range, azimuths and elevations. False (the default) will return the locations of the gate centers with no interpolation.

**filter\_transitions** [bool, optional] True to remove rays where the antenna was in transition between sweeps. False will include these rays. No rays will be removed if the `antenna_transition` attribute is not available (set to None).

#### Returns

**x, y, z** [2D array] Array containing the x, y and z, distances from the radar in meters for the center (or edges) for all gates in the sweep.

**get\_nyquist\_vel** (*self*, *sweep*, *check\_uniform=True*)

Return the Nyquist velocity in meters per second for a given sweep.

Raises a `LookupError` if the Nyquist velocity is not available, an `Exception` is raised if the velocities are not uniform in the sweep unless `check_uniform` is set to False.

#### Parameters

**sweep** [int] Sweep number to retrieve data for, 0 based.

**check\_uniform** [bool] True to check to perform a check on the Nyquist velocities that they are uniform in the sweep, False will skip this check and return the velocity of the first ray in the sweep.

#### Returns

**nyquist\_velocity** [float] Array containing the Nyquist velocity in m/s for a given sweep.

**get\_slice** (*self*, *sweep*)

Return a slice for selecting rays for a given sweep.

**get\_start** (*self*, *sweep*)

Return the starting ray index for a given sweep.

**get\_start\_end** (*self*, *sweep*)

Return the starting and ending ray for a given sweep.

**info** (*self*, *level='standard'*, *out=<\_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Print information on radar.

#### Parameters

**level** [{ 'compact', 'standard', 'full', 'c', 's', 'f' }] Level of information on radar object to print, compact is minimal information, standard more and full everything.

**out** [file-like] Stream to direct output to, default is to print information to standard out (the screen).

**init\_gate\_altitude** (*self*)

Initialize the `gate_altitude` attribute.

**init\_gate\_longitude\_latitude** (*self*)

Initialize or reset the `gate_longitude` and `gate_latitude` attributes.

**init\_gate\_x\_y\_z** (*self*)

Initialize or reset the `gate_{x, y, z}` attributes.

**init\_rays\_per\_sweep** (*self*)

Initialize or reset the `rays_per_sweep` attribute.



**iter\_azimuth** (*self*)  
Return an iterator which returns sweep azimuth data.

**iter\_elevation** (*self*)  
Return an iterator which returns sweep elevation data.

**iter\_end** (*self*)  
Return an iterator over the sweep end indices.

**iter\_field** (*self*, *field\_name*)  
Return an iterator which returns sweep field data.

**iter\_slice** (*self*)  
Return an iterator which returns sweep slice objects.

**iter\_start** (*self*)  
Return an iterator over the sweep start indices.

**iter\_start\_end** (*self*)  
Return an iterator over the sweep start and end indices.

`pyrad.io.read_data_mxpola.readCHRadData` (*filename*, *radar\_name*, *variableList*, *radial\_resolution*, *max\_range=inf*, *min\_range=0*)  
Reads a HDF5 file containing processed radar data in polar coordinates Parameters ——— filename: str  
complete path of the file

**radar\_name: str** name of MCH radar

**variableList: list** list of variables to be read

**radial\_resolution: float** resolution of the radar in metres (i.e. high: 83.3, low: 500.)

**max\_range: float** maximum range upto which to read data

**min\_range: float** minimum range from which to read data

**varPol: dict** the projected variables, the azimuth and the range

`pyrad.io.read_data_mxpola.readIDLRadData` (*filename*, *variableList*, *max\_range=inf*, *min\_range=0*)  
Reads a netcdf containing IDL processed radar data in polar coordinates Parameters ——— filename: str  
complete path of the file

**variableList: list** list of variables to be read

**varPol: dict** dictionary containing the variables, the azimuth and the range

**metadata: dict** dictionary containing the metadata of the file

`pyrad.io.read_data_mxpola.readMXPOLRadData` (*filename*, *variableList*, *max\_range=inf*, *min\_range=0*)  
Reads a netcdf containing processed radar data in polar coordinates Parameters ——— filename: str  
complete path of the file

**variableList: list** list of variables to be read

**varPol: dict** dictionary containing the variables, the azimuth and the range

**metadata: dict** dictionary containing the metadata of the file

`pyrad.io.read_data_mxpul.row_stack(a1, a2)`

Stacks data from subsequent sweeps, while padding “empty” columns from subsequent sweeps. Inputs —— a1:  
np.array

destination array

**a2: np.array** array which is added onto the first array

**out: np.array** stacked destination and additional array, with uniform shape

Created on Wed Dec 7 10:48:31 2016

@author: fvanden

Configuration file for mxpol pyart.core.Radar class. Some information may be redundant because this file is a copy from the ProfileLab toolkit.

**Functions to retrieve data from this file may be found in** `pyrad.io.read_data_mxpul` under the utilities section

## PYRAD.IO.READ\_DATA\_COSMO

Functions for reading COSMO data

<code>cosmo2radar_data(radar, cosmo_coord, cosmo_data)</code>	get the COSMO value corresponding to each radar gate using nearest neighbour interpolation
<code>cosmo2radar_coord(radar, cosmo_coord[, ...])</code>	Given the radar coordinates find the nearest COSMO model pixel
<code>get_cosmo_fields(cosmo_data, cosmo_ind[, ...])</code>	Get the COSMO data corresponding to each radar gate using a precomputed look up table of the nearest neighbour
<code>read_cosmo_data(fname[, field_names, celsius])</code>	Reads COSMO data from a netcdf file
<code>read_cosmo_coord(fname[, zmin])</code>	Reads COSMO coordinates from a netcdf file
<code>_ncvar_to_dict(ncvar[, dtype])</code>	Convert a NetCDF Dataset variable to a dictionary.
<code>_prepare_for_interpolation(x_radar, y_radar, ...)</code>	prepares the COSMO 3D volume for interpolation:
<code>_put_radar_in_swiss_coord(radar)</code>	puts the Cartesian grid of the radar coordinates in Swiss coordinates

`pyrad.io.read_data_cosmo._ncvar_to_dict(ncvar, dtype=<class 'numpy.float32'>)`

Convert a NetCDF Dataset variable to a dictionary.

`pyrad.io.read_data_cosmo._prepare_for_interpolation(x_radar, y_radar, z_radar, cosmo_coord, slice_xy=True, slice_z=False)`

**prepares the COSMO 3D volume for interpolation:**

1. if set slices the cosmo data to the area (or volume)

**covered by the radar**

2. creates the x, y, z grid for the interpolation

### Parameters

**x\_radar, y\_radar, z\_radar** [arrays] The Swiss coordinates of the radar

**cosmo\_coord** [dict] dictionary containing the COSMO coordinates

**slice\_xy** [boolean] if true the horizontal plane of the COSMO field is cut to the dimensions of the radar field

**slice\_z** [boolean] if true the vertical plane of the COSMO field is cut to the dimensions of the radar field

### Returns

**x\_cosmo, y\_cosmo, z\_cosmo** [1D arrays] arrays containing the flatten swiss coordinates of the COSMO data in the area of interest

**ind\_xmin, ind\_ymin, ind\_zmin, ind\_xmax, ind\_ymax, ind\_zmax** [ints] the minimum and maximum indices of each dimension

`pyrad.io.read_data_cosmo._put_radar_in_swiss_coord(radar)`

puts the Cartesian grid of the radar coordinates in Swiss coordinates

#### Parameters

**radar** [Radar] the radar object containing the information on the position of the radar gates

#### Returns

**x\_radar, y\_radar, z\_radar** [2D arrays] arrays containing swiss coordinates of the radar [in m]

`pyrad.io.read_data_cosmo.cosmo2radar_coord(radar, cosmo_coord, slice_xy=True, slice_z=False, field_name=None)`

Given the radar coordinates find the nearest COSMO model pixel

#### Parameters

**radar** [Radar] the radar object containing the information on the position of the radar gates

**cosmo\_coord** [dict] dictionary containing the COSMO coordinates

**slice\_xy** [boolean] if true the horizontal plane of the COSMO field is cut to the dimensions of the radar field

**slice\_z** [boolean] if true the vertical plane of the COSMO field is cut to the dimensions of the radar field

**field\_name** [str] name of the field

#### Returns

**cosmo\_ind\_field** [dict] dictionary containing a field of COSMO indices and metadata

`pyrad.io.read_data_cosmo.cosmo2radar_data(radar, cosmo_coord, cosmo_data, time_index=0, slice_xy=True, slice_z=False, field_names=['temperature'], dtype=<class 'numpy.float32'>)`

get the COSMO value corresponding to each radar gate using nearest neighbour interpolation

#### Parameters

**radar** [Radar] the radar object containing the information on the position of the radar gates

**cosmo\_coord** [dict] dictionary containing the COSMO coordinates

**cosmo\_data** [dict] dictionary containing the COSMO data

**time\_index** [int] index of the forecasted data

**slice\_xy** [boolean] if true the horizontal plane of the COSMO field is cut to the dimensions of the radar field

**slice\_z** [boolean] if true the vertical plane of the COSMO field is cut to the dimensions of the radar field

**field\_names** [str] names of COSMO fields to convert (default temperature)

**dtype** [numpy data type object] the data type of the output data

#### Returns

**cosmo\_fields** [list of dict] list of dictionary with the COSMO fields and metadata

```
pyrad.io.read_data_cosmo.get_cosmo_fields(cosmo_data, cosmo_ind, time_index=0,  
                                           field_names=['temperature'])
```

Get the COSMO data corresponding to each radar gate using a precomputed look up table of the nearest neighbour

#### Parameters

**cosmo\_data** [dict] dictionary containing the COSMO data and metadata

**cosmo\_ind** [dict] dictionary containing a field of COSMO indices and metadata

**time\_index** [int] index of the forecasted data

**field\_names** [str] names of COSMO parameters (default temperature)

#### Returns

**cosmo\_fields** [list of dict] dictionary with the COSMO fields and metadata

```
pyrad.io.read_data_cosmo.read_cosmo_coord(fname, zmin=None)
```

Reads COSMO coordinates from a netcdf file

#### Parameters

**fname** [str] name of the file to read

#### Returns

**cosmo\_coord** [dictionary] dictionary with the data and metadata

```
pyrad.io.read_data_cosmo.read_cosmo_data(fname, field_names=['temperature'], cel-  
                                          sious=True)
```

Reads COSMO data from a netcdf file

#### Parameters

**fname** [str] name of the file to read

**field\_names** [str] name of the variable to read

**celsius** [Boolean] if True and variable temperature converts data from Kelvin to Centigrade

#### Returns

**cosmo\_data** [dictionary] dictionary with the data and metadata



## PYRAD.IO.READ\_DATA\_HZT

Functions for reading HZT data

<code>hzt2radar_data(radar, hzt_coord, hzt_data[, ...])</code>	get the HZT value corresponding to each radar gate using nearest neighbour interpolation
<code>hzt2radar_coord(radar, hzt_coord[, ...])</code>	Given the radar coordinates find the nearest HZT pixel
<code>get_iso0_field(hzt_data, hzt_ind, z_radar[, ...])</code>	Get the height over iso0 data corresponding to each radar gate using a precomputed look up table of the nearest neighbour
<code>read_hzt_data(fname[, chy0, chx0, read_lib])</code>	Reads iso-0 degree data from an HZT file
<code>_prepare_for_interpolation(x_radar, y_radar, ...)</code>	prepares the HZT 2D volume for interpolation:

```
pyrad.io.read_data_hzt._prepare_for_interpolation(x_radar, y_radar, hzt_coord,
                                                    slice_xy=True)
```

**prepares the HZT 2D volume for interpolation:**

1. if set slices the cosmo data to the area covered by the radar
2. creates the x, y grid for the interpolation

### Parameters

**x\_radar, y\_radar** [arrays] The Swiss coordinates of the radar

**hzt\_coord** [dict] dictionary containing the HZT coordinates

**slice\_xy** [boolean] if true the horizontal plane of the HZT field is cut to the dimensions of the radar field

### Returns

**x\_hzt, y\_hzt** [1D arrays] arrays containing the flatten swiss coordinates of the HZT data in the area of interest [m]

**ind\_xmin, ind\_ymin, ind\_xmax, ind\_ymax** [ints] the minimum and maximum indices of each dimension

```
pyrad.io.read_data_hzt.get_iso0_field(hzt_data, hzt_ind, z_radar,
                                       field_name='height_over_iso0')
```

Get the height over iso0 data corresponding to each radar gate using a precomputed look up table of the nearest neighbour

### Parameters

**hzt\_data** [dict] dictionary containing the HZT data and metadata

**hzt\_ind** [dict] dictionary containing a field of HZT indices and metadata

**z\_radar** [ndarray] gates altitude [m MSL]

**field\_name** [str] names of HZT parameters (default height\_over\_iso0)

#### Returns

**iso0\_field** [list of dict] dictionary with the height over iso0 field and metadata

```
pyrad.io.read_data_hzt.hzt2radar_coord(radar,          hzt_coord,          slice_xy=True,
                                         field_name=None)
```

Given the radar coordinates find the nearest HZT pixel

#### Parameters

**radar** [Radar] the radar object containing the information on the position of the radar gates

**hzt\_coord** [dict] dictionary containing the HZT coordinates

**slice\_xy** [boolean] if true the horizontal plane of the HZT field is cut to the dimensions of the radar field

**field\_name** [str] name of the field

#### Returns

**hzt\_ind\_field** [dict] dictionary containing a field of HZT indices and metadata

```
pyrad.io.read_data_hzt.hzt2radar_data(radar,  hzt_coord,  hzt_data,  slice_xy=True,
                                         field_name='height_over_iso0')
```

get the HZT value corresponding to each radar gate using nearest neighbour interpolation

#### Parameters

**radar** [Radar] the radar object containing the information on the position of the radar gates

**hzt\_coord** [dict] dictionary containing the HZT coordinates

**hzt\_data** [dict] dictionary containing the HZT data

**slice\_xy** [boolean] if true the horizontal plane of the COSMO field is cut to the dimensions of the radar field

**field\_name** [str] name of HZT fields to convert (default height\_over\_iso0)

#### Returns

**hzt\_fields** [list of dict] list of dictionary with the HZT fields and metadata

```
pyrad.io.read_data_hzt.read_hzt_data(fname, chy0=255.0, chx0=-160.0, read_lib='C')
```

Reads iso-0 degree data from an HZT file

#### Parameters

**fname** [str] name of the file to read

**chy0, chx0: float** south west point of grid in Swiss coordinates [km]

**read\_lib** [str] Type of METRANET read library used. Can be 'C' or 'python'

#### Returns

**hzt\_data** [dictionary] dictionary with the data and metadata



## PYRAD.IO.READ\_DATA\_DEM

Functions for reading data derived from Digital Elevation Models (DEM)

<code>dem2radar_data(radar, dem_data[, slice_xy, ...])</code>	get the DEM value corresponding to each radar gate using nearest neighbour interpolation
<code>read_idrisi_data(fname, field_name[, fill_value])</code>	Reads DEM data from an IDRISI .rst file
<code>read_idrisi_metadata(fname)</code>	Reads DEM metadata from a IDRISI .rdc file
<code>_prepare_for_interpolation(x_radar, y_radar, ...)</code>	prepares the DEM 2D volume for interpolation:

`pyrad.io.read_data_dem._prepare_for_interpolation(x_radar, y_radar, dem_coord, slice_xy=True)`

**prepares the DEM 2D volume for interpolation:**

1. if set slices the DEM data to the area

**covered by the radar**

2. creates the x, y grid for the interpolation

### Parameters

**x\_radar, y\_radar** [arrays] The Swiss coordinates of the radar

**dem\_coord** [dict] dictionary containing the DEM coordinates

**slice\_xy** [boolean] if true the horizontal plane of the DEM field is cut to the dimensions of the radar field

### Returns

**x\_dem, y\_dem** [1D arrays] arrays containing the flatten swiss coordinates of the DEM data in the area of interest

**ind\_xmin, ind\_ymin, ind\_xmax, ind\_ymax** [ints] the minimum and maximum indices of each dimension

`pyrad.io.read_data_dem.dem2radar_data(radar, dem_data, slice_xy=True, field_name='visibility')`  
get the DEM value corresponding to each radar gate using nearest neighbour interpolation

### Parameters

**radar** [Radar] the radar object containing the information on the position of the radar gates

**dem\_data** [dict] dictionary containing the DEM data

**slice\_xy** [boolean] if true the horizontal plane of the DEM field is cut to the dimensions of the radar field

**field\_names** [str] names of DEM fields to convert

**Returns**

**dem\_field** [dict] Dictionary with the DEM fields and metadata

`pyrad.io.read_data_dem.read_idrиси_data` (*fname, field\_name, fill\_value=-99.0*)

Reads DEM data from an IDRISI .rst file

**Parameters**

**fname** [str] name of the file to read

**field\_name** [str] name of the readed variable

**fill\_value** [float] The fill value

**Returns**

**dem\_data** [dictionary] dictionary with the data and metadata

`pyrad.io.read_data_dem.read_idrиси_metadata` (*fname*)

Reads DEM metadata from a IDRISI .rdc file

**Parameters**

**fname** [str] name of the file to read

**Returns**

**metadata** [dictionary] dictionary with the metadata

## PYRAD.IO.READ\_DATA\_SENSOR

Functions for reading data from other sensors

<i>read_windmills_data</i> (fname)	Read the wind mills data csv file
<i>read_thundertracking_info</i> (fname)	Reads the TRT info used for thundertracking
<i>read_trt_info_all</i> (info_path)	Reads all the TRT info files
<i>read_trt_info_all2</i> (info_path)	Reads all the TRT info files
<i>read_trt_info</i> (fname)	Reads the TRT info used for thundertracking and contained in a text file.
<i>read_trt_info2</i> (fname)	Reads the TRT info used for thundertracking and contained in a text file.
<i>read_trt_scores</i> (fname)	Reads the TRT scores contained in a text file.
<i>read_trt_cell_lightning</i> (fname)	Reads the lightning data of a TRT cell.
<i>read_trt_data</i> (fname)	Reads the TRT data contained in a text file.
<i>read_trt_traj_data</i> (fname)	Reads the TRT cell data contained in a text file.
<i>read_trt_thundertracking_traj_data</i> (fname)	Reads the TRT cell data contained in a text file.
<i>read_lightning</i> (fname[, filter_data])	Reads lightning data contained in a text file.
<i>read_meteorage</i> (fname)	Reads METEORAGE lightning data contained in a text file.
<i>read_lightning_traj</i> (fname)	Reads lightning trajectory data contained in a csv file.
<i>read_lightning_all</i> (fname[, labels])	Reads a file containing lightning data and co-located polarimetric data.
<i>get_sensor_data</i> (date, datatype, cfg)	Gets data from a point measurement sensor (rain gauge or disdrometer)
<i>read_smn</i> (fname)	Reads SwissMetNet data contained in a csv file
<i>read_smn2</i> (fname)	Reads SwissMetNet data contained in a csv file with format station,time,value
<i>read_disdro_scattering</i> (fname)	Reads scattering parameters computed from disdrometer data contained in a text file
<i>read_disdro</i> (fname)	Reads scattering parameters computed from disdrometer data contained in a text file

`pyrad.io.read_data_sensor.get_sensor_data` (*date*, *datatype*, *cfg*)  
Gets data from a point measurement sensor (rain gauge or disdrometer)

### Parameters

- date** [datetime object] measurement date
- datatype** [str] name of the data type to read
- cfg** [dictionary] dictionary containing sensor information

**Returns**

**sensordate , sensorvalue, label, period** [tuple] date, value, type of sensor and measurement period

`pyrad.io.read_data_sensor.read_disdro (fname)`

Reads scattering parameters computed from disdrometer data contained in a text file

**Parameters**

**fname** [str] path of time series file

**Returns**

**date, precipitype, variable, scattering temperature: tuple** The read values

`pyrad.io.read_data_sensor.read_disdro_scattering (fname)`

Reads scattering parameters computed from disdrometer data contained in a text file

**Parameters**

**fname** [str] path of time series file

**Returns**

**date, precipitype, lwc, rr, zh, zv, zdr, ldr, ah, av, adiff, kdp, deltaco, rhohv** [tuple] The read values

`pyrad.io.read_data_sensor.read_lightning (fname, filter_data=True)`

Reads lightning data contained in a text file. The file has the following fields:

flashnr: (0 is for noise) UTC seconds of the day Time within flash (in seconds) Latitude (decimal degrees) Longitude (decimal degrees) Altitude (m MSL) Power (dBm)

**Parameters**

**fname** [str] path of time series file

**filter\_data** [Boolean] if True filter noise (flashnr = 0)

**Returns**

**flashnr, time\_data, time\_in\_flash, lat, lon, alt, dBm** [tuple] A tuple containing the read values. None otherwise

`pyrad.io.read_data_sensor.read_lightning_all (fname, labels=['hydro [-]', 'KDPc [deg/Km]', 'dBZc [dBZ]', 'RhoHvc [-]', 'TEMP [deg C]', 'ZDRc [dB]'])`

Reads a file containing lightning data and co-located polarimetric data. fields:

flashnr time data Time within flash (in seconds) Latitude (decimal degrees) Longitude (decimal degrees) Altitude (m MSL) Power (dBm) Polarimetric values at flash position

**Parameters**

**fname** [str] path of time series file

**labels** [list of str] The polarimetric variables labels

**Returns**

**flashnr, time\_data, time\_in\_flash, lat, lon, alt, dBm,**

**pol\_vals\_dict** [tuple] A tuple containing the read values. None otherwise

`pyrad.io.read_data_sensor.read_lightning_traj(fname)`

Reads lightning trajectory data contained in a csv file. The file has the following fields:

Date UTC [seconds since midnight] # Flash Flash Power (dBm) Value at flash Mean value in a 3x3x3 polar box Min value in a 3x3x3 polar box Max value in a 3x3x3 polar box # valid values in the polar box

**Parameters**

**fname** [str] path of time series file

**Returns**

**time\_flash, flashnr, dBm, val\_at\_flash, val\_mean, val\_min, val\_max,**

**nval** [tuple] A tuple containing the read values. None otherwise

`pyrad.io.read_data_sensor.read_meteorage(fname)`

Reads METEORAGE lightning data contained in a text file. The file has the following fields:

date: date + time + time zone lon: longitude [degree] lat: latitude [degree] intens: amplitude [kilo amperes] ns: number of strokes of the flash mode: kind of localization [0,15] intra: 1 = intra-cloud , 0 = cloud-to-ground ax: length of the semi-major axis of the ellipse [km] ki2: standard deviation on the localization computation ( $K_i^2$ ) ecc: eccentricity (major-axis / minor-axis) incl: ellipse inclination (angle with respect to the North, +90° is

East) [degrees]

sind: stroke index within the flash

**Parameters**

**fname** [str] path of time series file

**Returns**

**stroke\_time, lon, lat, intens, ns, mode, intra, ax, ki2, ecc, incl,**

**sind** [tuple] A tuple containing the read values. None otherwise

`pyrad.io.read_data_sensor.read_smn(fname)`

Reads SwissMetNet data contained in a csv file

**Parameters**

**fname** [str] path of time series file

**Returns**

**smn\_id, date , pressure, temp, rh, precip, wspeed, wdir** [tuple] The read values

`pyrad.io.read_data_sensor.read_smn2(fname)`

Reads SwissMetNet data contained in a csv file with format station,time,value

**Parameters**

**fname** [str] path of time series file

**Returns**

**smn\_id, date , value** [tuple] The read values

`pyrad.io.read_data_sensor.read_thundertracking_info(fname)`

Reads the TRT info used for thundertracking

**Parameters**

**fname** [str] Name of the file containing the info

**Returns**

**A tuple containing the read values. None otherwise. The read values are**

**id, max\_rank, nscans\_Xband, time\_start, time\_end**

`pyrad.io.read_data_sensor.read_trt_cell_lightning(fname)`

Reads the lightning data of a TRT cell. The file has the following fields:

traj\_ID yyyyymmddHHMM lon lat area RANKr nflashes flash\_dens

**Parameters**

**fname** [str] path of the TRT data file

**Returns**

**A tuple containing the read values. None otherwise**

`pyrad.io.read_data_sensor.read_trt_data(fname)`

Reads the TRT data contained in a text file. The file has the following fields:

traj\_ID yyyyymmddHHMM

Description of ellipsis: lon [deg] lat [deg] ell\_L [km] long ell\_S [km] short ell\_or [deg] orientation area [km2]

Cell speed: vel\_x [km/h] vel\_y [km/h] det [dBZ]: detection threshold RANKr from 0 to 40 (int)

Lightning information: CG- number (int) CG+ number (int) CG number (int) %CG+ [%]

Echo top information: ET45 [km] echotop 45 max ET45m [km] echotop 45 median ET15 [km] echotop 15 max ET15m [km] echotop 15 median

VIL and max echo: VIL [kg/m2] vertical integrated liquid content maxH [km] height of maximum reflectivity (maximum on the cell) maxHm [km] height of maximum reflectivity (median per cell)

POH [%] RANK (deprecated)

standard deviation of the current time step cell velocity respect to the previous time: Dvel\_x [km/h] Dvel\_y [km/h]

cell\_contour\_lon-lat

**Parameters**

**fname** [str] path of the TRT data file

**Returns**

**A tuple containing the read values. None otherwise**

`pyrad.io.read_data_sensor.read_trt_info(fname)`

Reads the TRT info used for thundertracking and contained in a text file.

**Parameters**

**fname** [str] path of the TRT info file

**Returns**

**A tuple containing the read values. None otherwise. The read values are**

**trt\_time, id, rank, nscans, azi, rng, lat, lon, ell\_l, ell\_s, ell\_or,  
vel\_x, vel\_y, det**

`pyrad.io.read_data_sensor.read_trt_info2(fname)`

Reads the TRT info used for thundertracking and contained in a text file.

**Parameters**

**fname** [str] path of the TRT info file

**Returns**

**A tuple containing the read values. None otherwise. The read values are**

**trt\_time, id, rank, scan\_time, azi, rng, lat, lon, ell\_l, ell\_s, ell\_or,  
vel\_x, vel\_y, det**

`pyrad.io.read_data_sensor.read_trt_info_all(info_path)`

Reads all the TRT info files

**Parameters**

**info\_path** [str] directory where the files are stored

**Returns**

**A tuple containing the read values. None otherwise. The read values are**

**trt\_time, id, rank, nscans, azi, rng, lat, lon, ell\_l, ell\_s, ell\_or,  
vel\_x, vel\_y, det**

`pyrad.io.read_data_sensor.read_trt_info_all2(info_path)`

Reads all the TRT info files

**Parameters**

**info\_path** [str] directory where the files are stored

**Returns**

**A tuple containing the read values. None otherwise. The read values are**

**trt\_time, id, rank, scan\_time, azi, rng, lat, lon, ell\_l, ell\_s, ell\_or,  
vel\_x, vel\_y, det**

`pyrad.io.read_data_sensor.read_trt_scores(fname)`

Reads the TRT scores contained in a text file. The file has the following fields:

traj ID max flash density time max flash density rank max flash density max rank time max rank

**Parameters**

**fname** [str] path of the TRT data file

**Returns**

**A tuple containing the read values. None otherwise**

`pyrad.io.read_data_sensor.read_trt_thundertracking_traj_data(fname)`

Reads the TRT cell data contained in a text file. The file has the following fields:

traj\_ID scan\_ordered\_time scan\_time azi rng yyyyymmddHHMM  
lon [deg] lat [deg] ell\_L [km] long ell\_S [km] short ell\_or [deg] orientation area [km2]  
vel\_x [km/h] cell speed vel\_y [km/h] det [dBZ] detection threshold RANKr from 0 to 40 (int)  
CG- number (int) CG+ number (int) CG number (int) %CG+ [%]  
ET45 [km] echotop 45 max ET45m [km] echotop 45 median ET15 [km] echotop 15 max ET15m [km] echotop 15 median VIL [kg/m2] vertical integrated liquid content maxH [km] height of maximum reflectivity (maximum on the cell) maxHm [km] height of maximum reflectivity (median per cell) POH [%] RANK (deprecated)  
Standard deviation of the current time step cell velocity respect to the previous time: Dvel\_x [km/h] Dvel\_y [km/h]  
cell\_contour\_lon-lat

**Parameters**

**fname** [str] path of the TRT data file

**Returns**

**A tuple containing the read values. None otherwise**

`pyrad.io.read_data_sensor.read_trt_traj_data(fname)`

Reads the TRT cell data contained in a text file. The file has the following fields:

traj\_ID yyyyymmddHHMM  
lon [deg] lat [deg] ell\_L [km] long ell\_S [km] short ell\_or [deg] orientation area [km2]  
vel\_x [km/h] cell speed vel\_y [km/h] det [dBZ] detection threshold RANKr from 0 to 40 (int)  
CG- number (int) CG+ number (int) CG number (int) %CG+ [%]  
ET45 [km] echotop 45 max ET45m [km] echotop 45 median ET15 [km] echotop 15 max ET15m [km] echotop 15 median VIL [kg/m2] vertical integrated liquid content maxH [km] height of maximum reflectivity (maximum on the cell) maxHm [km] height of maximum reflectivity (median per cell) POH [%] RANK (deprecated)  
Standard deviation of the current time step cell velocity respect to the previous time: Dvel\_x [km/h] Dvel\_y [km/h]  
cell\_contour\_lon-lat

**Parameters**

**fname** [str] path of the TRT data file

**Returns**

**A tuple containing the read values. None otherwise**

`pyrad.io.read_data_sensor.read_windmills_data(fname)`

Read the wind mills data csv file

**Parameters**

**fname** [str] path of the windmill data file

**Returns**

**windmill\_dict** [dict] A dictionary containing all the parameters or None



## PYRAD.IO.READ\_DATA\_SUN

Functions for reading data used in sun monitoring

<code>read_sun_hits_multiple_days(cfg, time_ref[, ...])</code>	Reads sun hits data from multiple file sources
<code>read_sun_hits(fname)</code>	Reads sun hits data contained in a csv file
<code>read_sun_retrieval(fname)</code>	Reads sun retrieval data contained in a csv file
<code>read_solar_flux(fname)</code>	Reads solar flux data from the DRAO observatory in Canada

`pyrad.io.read_data_sun.read_solar_flux(fname)`  
Reads solar flux data from the DRAO observatory in Canada

### Parameters

**fname** [str] path of time series file

### Returns

**flux\_datetime** [datetime array] the date and time of the solar flux retrievals

**flux\_value** [array] the observed solar flux

`pyrad.io.read_data_sun.read_sun_hits(fname)`  
Reads sun hits data contained in a csv file

### Parameters

**fname** [str] path of time series file

### Returns

**date, ray, nrng, rad\_el, rad\_az, sun\_el, sun\_az, ph, ph\_std, npv, nvalh,**

**pv, pv\_std, npv, nvalv, zdr, zdr\_std, nzdr, nvalzdr** [tuple] Each parameter is an array containing a time series of information on a variable

`pyrad.io.read_data_sun.read_sun_hits_multiple_days(cfg, time_ref, nfiles=1)`  
Reads sun hits data from multiple file sources

### Parameters

**cfg** [dict] dictionary with configuration data to find out the right file

**time\_ref** [datetime object] reference time

**nfiles** [int] number of files to read

### Returns

**date, ray, nrng, rad\_el, rad\_az, sun\_el, sun\_az, ph, ph\_std, nph, nvalh,**  
**pv, pv\_std, npv, nvalv, zdr, zdr\_std, nzdr, nvalzdr** [tuple] Each parameter is an array containing a time series of information on a variable

`pyrad.io.read_data_sun.read_sun_retrieval` (*fname*)

Reads sun retrieval data contained in a csv file

#### Parameters

**fname** [str] path of time series file

#### Returns

**first\_hit\_time, last\_hit\_time, nhits\_h, el\_width\_h, az\_width\_h, el\_bias\_h,**  
**az\_bias\_h, dBm\_sun\_est, std\_dBm\_sun\_est, sf\_h,**  
**nhits\_v, el\_width\_v, az\_width\_v, el\_bias\_v, az\_bias\_v, dBmv\_sun\_est,**  
**std\_dBmv\_sun\_est, sf\_v,**  
**nhits\_zdr, zdr\_sun\_est, std\_zdr\_sun\_est,**  
**sf\_ref, ref\_time** [tuple] Each parameter is an array containing a time series of information on a variable

## PYRAD.IO.READ\_DATA\_OTHER

Functions for reading auxiliary data

<i>read_proc_periods</i> (fname)	Reads a file containing the start and stop times of periods to process
<i>read_profile_ts</i> (fname_list, labels[, hres, ...])	Reads a collection of profile data file and creates a time series
<i>read_histogram_ts</i> (fname_list, datatype[, t_res])	Reads a collection of histogram data file and creates a time series
<i>read_quantiles_ts</i> (fname_list[, step, qmin, ...])	Reads a collection of quantiles data file and creates a time series
<i>read_rhi_profile</i> (fname[, labels])	Reads a monitoring time series contained in a csv file
<i>read_last_state</i> (fname)	Reads a file containing the date of acquisition of the last volume processed
<i>read_status</i> (voltime, cfg[, ind_rad])	Reads rad4alp xml status file.
<i>read_rad4alp_cosmo</i> (fname, datatype[, ngates])	Reads rad4alp COSMO data binary file.
<i>read_rad4alp_vis</i> (fname, datatype)	Reads rad4alp visibility data binary file.
<i>read_histogram</i> (fname)	Reads a histogram contained in a csv file
<i>read_quantiles</i> (fname)	Reads quantiles contained in a csv file
<i>read_excess_gates</i> (fname)	Reads a csv files containing the position of gates exceeding a given percentile of frequency of occurrence
<i>read_colocated_gates</i> (fname)	Reads a csv files containing the position of colocated gates
<i>read_colocated_data</i> (fname)	Reads a csv files containing colocated data
<i>read_colocated_data_time_avg</i> (fname)	Reads a csv files containing time averaged colocated data
<i>read_timeseries</i> (fname)	Reads a time series contained in a csv file
<i>read_ts_cum</i> (fname)	Reads a time series of precipitation accumulation contained in a csv file
<i>read_ml_ts</i> (fname)	Reads a melting layer time series contained in a csv file
<i>read_monitoring_ts</i> (fname[, sort_by_date])	Reads a monitoring time series contained in a csv file
<i>read_monitoring_ts_old</i> (fname)	Reads an old format of the monitoring time series contained in a text file
<i>read_intercomp_scores_ts</i> (fname[, sort_by_date])	Reads a radar intercomparison scores csv file
<i>read_intercomp_scores_ts_old</i> (fname)	Reads a radar intercomparison scores csv file in old format
<i>read_intercomp_scores_ts_old_v0</i> (fname[, ...])	Reads a radar intercomparison scores csv file in the oldest format

Continued on next page

Table 1 – continued from previous page

<code>read_selfconsistency(fname)</code>	Reads a self-consistency table with Zdr, Kdp/Zh columns
<code>read_antenna_pattern(fname[, linear, twoway])</code>	Read antenna pattern from file

`pyrad.io.read_data_other.read_antenna_pattern(fname, linear=False, twoway=False)`  
Read antenna pattern from file

**Parameters**

**fname** [str] path of the antenna pattern file  
**linear** [boolean] if true the antenna pattern is given in linear units  
**twoway** [boolean] if true the attenuation is two-way

**Returns**

**pattern** [dict] dictionary with the fields angle and attenuation

`pyrad.io.read_data_other.read_colocated_data(fname)`  
Reads a csv files containing colocated data

**Parameters**

**fname** [str] path of time series file

**Returns**

**rad1\_time, rad1\_ray\_ind, rad1\_rng\_ind, rad1\_ele, rad1\_azi, rad1\_rng,**  
**rad1\_val, rad2\_time, rad2\_ray\_ind, rad2\_rng\_ind, rad2\_ele, rad2\_azi,**  
**rad2\_rng, rad2\_val** [tuple] A tuple with the data read. None otherwise

`pyrad.io.read_data_other.read_colocated_data_time_avg(fname)`  
Reads a csv files containing time averaged colocated data

**Parameters**

**fname** [str] path of time series file

**Returns**

**rad1\_time, rad1\_ray\_ind, rad1\_rng\_ind, rad1\_ele, rad1\_azi, rad1\_rng,**  
**rad1\_val, rad2\_time, rad2\_ray\_ind, rad2\_rng\_ind, rad2\_ele, rad2\_azi,**  
**rad2\_rng, rad2\_val** [tuple] A tuple with the data read. None otherwise

`pyrad.io.read_data_other.read_colocated_gates(fname)`  
Reads a csv files containing the position of colocated gates

**Parameters**

**fname** [str] path of time series file

**Returns**

**rad1\_ray\_ind, rad1\_rng\_ind, rad1\_ele, rad1\_azi, rad1\_rng,**  
**rad2\_ray\_ind, rad2\_rng\_ind, rad2\_ele, rad2\_azi, rad2\_rng** [tuple] A tuple with the  
data read. None otherwise

`pyrad.io.read_data_other.read_excess_gates(fname)`  
Reads a csv files containing the position of gates exceeding a given percentile of frequency of occurrence

**Parameters**

**fname** [str] path of time series file

**Returns**

**rad1\_ray\_ind, rad1\_rng\_ind, rad1\_ele, rad1\_azi, rad1\_rng,**

**rad2\_ray\_ind, rad2\_rng\_ind, rad2\_ele, rad2\_azi, rad2\_rng** [tuple] A tuple with the data read. None otherwise

`pyrad.io.read_data_other.read_histogram(fname)`

Reads a histogram contained in a csv file

**Parameters**

**fname** [str] path of time series file

**Returns**

**hist, bin\_edges** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_histogram_ts(fname_list, datatype, t_res=300.0)`

Reads a collection of histogram data file and creates a time series

**Parameters**

**fname\_list** [str] list of files to read

**datatype** [str] The data type (dBZ, ZDR, etc.)

**t\_res** [float] time resolution [s]. If None the time resolution is taken as the median

**Returns**

**tbin\_edges, bin\_edges, data\_ma, datetime\_arr** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_intercomp_scores_ts(fname, sort_by_date=False)`

Reads a radar intercomparison scores csv file

**Parameters**

**fname** [str] path of time series file

**sort\_by\_date** [bool] if True, the read data is sorted by date prior to exit

**Returns**

**date\_vec, np\_vec, meanbias\_vec, medianbias\_vec, quant25bias\_vec,**

**quant75bias\_vec, modebias\_vec, corr\_vec, slope\_vec, intercep\_vec,**

**intercep\_slope1\_vec** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_intercomp_scores_ts_old(fname)`

Reads a radar intercomparison scores csv file in old format

**Parameters**

**fname** [str] path of time series file

**Returns**

**date\_vec, np\_vec, meanbias\_vec, medianbias\_vec, quant25bias\_vec,**

**quant75bias\_vec, modebias\_vec, corr\_vec, slope\_vec, intercep\_vec,**

**intercep\_slope1\_vec** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_intercomp_scores_ts_old_v0(fname, corr_min=0.6,  
np_min=9)`

Reads a radar intercomparison scores csv file in the oldest format

**Parameters**

**fname** [str] path of time series file

**Returns**

**date\_vec, np\_vec, meanbias\_vec, medianbias\_vec, quant25bias\_vec, quant75bias\_vec, modebias\_vec, corr\_vec, slope\_vec, intercep\_vec, intercep\_slope1\_vec** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_last_state(fname)`

Reads a file containing the date of acquisition of the last volume processed

**Parameters**

**fname** [str] name of the file to read

**Returns**

**last\_state** [datetime object] the date

`pyrad.io.read_data_other.read_ml_ts(fname)`

Reads a melting layer time series contained in a csv file

**Parameters**

**fname** [str] path of time series file

**Returns**

**dt\_ml, ml\_top\_avg, ml\_top\_std, thick\_avg, thick\_std, nrays\_valid, nrays\_total** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_monitoring_ts(fname, sort_by_date=False)`

Reads a monitoring time series contained in a csv file

**Parameters**

**fname** [str] path of time series file

**sort\_by\_date** [bool] if True, the read data is sorted by date prior to exit

**Returns**

**date, np\_t, central\_quantile, low\_quantile, high\_quantile** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_monitoring_ts_old(fname)`

Reads an old format of the monitoring time series contained in a text file

**Parameters**

**fname** [str] path of time series file

**Returns**

**date, np\_t, central\_quantile, low\_quantile, high\_quantile** [tuple] The read data in the current format. None otherwise

`pyrad.io.read_data_other.read_proc_periods(fname)`

Reads a file containing the start and stop times of periods to process

**Parameters**

**fname** [str] name of the file to read

**Returns**

**starttimes, endtimes** [array of datetime objects or None] The start and end times of the periods to process if the reading has been successful

`pyrad.io.read_data_other.read_profile_ts` (*fname\_list*, *labels*, *hres=None*, *label\_nr=0*, *t\_res=300.0*)

Reads a collection of profile data file and creates a time series

**Parameters**

**fname\_list** [str] list of files to read

**labels** [list of str] The data labels

**hres** [float] Height resolution

**label\_nr** [int] the label nr of the data that will be used in the time series

**t\_res** [float] time resolution [s]. If None the time resolution is taken as the median

**Returns**

**tbin\_edges, hbin\_edges, np\_ma, data\_ma, datetime\_arr** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_quantiles` (*fname*)

Reads quantiles contained in a csv file

**Parameters**

**fname** [str] path of time series file

**Returns**

**quantiles, values** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_quantiles_ts` (*fname\_list*, *step=5.0*, *qmin=0.0*, *qmax=100.0*, *t\_res=300.0*)

Reads a collection of quantiles data file and creates a time series

**Parameters**

**fname\_list** [str] list of files to read

**step, qmin, qmax** [float] The minimum, maximum and step quantiles

**t\_res** [float] time resolution [s]. If None the time resolution is taken as the median

**Returns**

**tbin\_edges, qbin\_edges, data\_ma, datetime\_arr** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_rad4alp_cosmo` (*fname*, *datatype*, *ngates=0*)

Reads rad4alp COSMO data binary file.

**Parameters**

**fname** [str] name of the file to read

**datatype** [str] name of the data type

**ngates** [int] maximum number of range gates per ray. If larger than 0 the radar field will be cut accordingly.

**Returns**

**field** [dictionary] The data field

`pyrad.io.read_data_other.read_rad4alp_vis` (*fname*, *datatype*)

Reads rad4alp visibility data binary file.

**Parameters**

**fname** [str] name of the file to read

**datatype** [str] name of the data type

**Returns**

**field\_list** [list of dictionaries] A data field. Each element of the list corresponds to one elevation

`pyrad.io.read_data_other.read_rhi_profile` (*fname*, *labels*=['50.0-percentile', '25.0-percentile', '75.0-percentile'])

Reads a monitoring time series contained in a csv file

**Parameters**

**fname** [str] path of time series file

**labels** [list of str] The data labels

**Returns**

**height, np\_t, vals** [tuple] The read data. None otherwise

`pyrad.io.read_data_other.read_selfconsistency` (*fname*)

Reads a self-consistency table with Zdr, Kdp/Zh columns

**Parameters**

**fname** [str] path of time series file

**Returns**

**zdr, kdpzh** [arrays] The read values

`pyrad.io.read_data_other.read_status` (*voltime*, *cfg*, *ind\_rad=0*)

Reads rad4alp xml status file.

**Parameters**

**voltime** [datetime object] volume scan time

**cfg: dictionary of dictionaries** configuration info to figure out where the data is

**ind\_rad: int** radar index

**Returns**

**root** [root element object] The information contained in the status file

`pyrad.io.read_data_other.read_timeseries` (*fname*)

Reads a time series contained in a csv file

**Parameters**

**fname** [str] path of time series file

**Returns**

**date, value** [tuple] A datetime object array containing the time and a numpy masked array containing the value. None otherwise

`pyrad.io.read_data_other.read_ts_cum` (*fname*)

Reads a time series of precipitation accumulation contained in a csv file



**Parameters**

**fname** [str] path of time series file

**Returns**

**date, np\_radar, radar\_value, np\_sensor, sensor\_value** [tuple] The data read



## PYRAD.IO.WRITE\_DATA

Functions for writing pyrad output data

<i>write_proc_periods</i> (start_times, end_times, fname)	writes an output file containing start and stop times of periods to process
<i>write_fixed_angle</i> (time_data, fixed_angle, ...)	writes an output file with the fixed angle data
<i>write_ts_lightning</i> (flashnr, time_data, ...)	writes the LMA sources data and the value of the collocated polarimetric variables
<i>send_msg</i> (sender, receiver_list, subject, fname)	sends the content of a text file by email
<i>write_alarm_msg</i> (radar_name, param_name_unit, ...)	writes an alarm file
<i>write_last_state</i> (datetime_last, fname)	writes SwissMetNet data in format datetime,avg_value, std_value
<i>write_smn</i> (datetime_vec, value_avg_vec, ...)	writes SwissMetNet data in format datetime,avg_value, std_value
<i>write_trt_info</i> (ids, max_rank, nscans, ...)	writes TRT info of the thundertracking
<i>write_trt_cell_data</i> (traj_ID, yyyymmddHHMM, ...)	writes TRT cell data
<i>write_trt_thundertracking_data</i> (traj_ID, ...)	writes TRT cell data of the thundertracking scan
<i>write_trt_cell_scores</i> (traj_ID, ...)	writes TRT cells scores
<i>write_trt_cell_lightning</i> (cell_ID, cell_time, ...)	writes the lightning data for each TRT cell
<i>write_trt_rpc</i> (cell_ID, cell_time, lon, lat, ...)	writes the rimed particles column data for a TRT cell
<i>write_rhi_profile</i> (hvec, data, nvalid_vec, ...)	writes the values of an RHI profile in a text file
<i>write_field_coverage</i> (quantiles, values, ...)	writes the quantiles of the coverage on a particular sector
<i>write_cdf</i> (quantiles, values, ntot, nnan, ...)	writes a cumulative distribution function
<i>write_histogram</i> (bin_edges, values, fname[, ...])	writes a histogram
<i>write_quantiles</i> (quantiles, values, fname[, ...])	writes quantiles
<i>write_ts_polar_data</i> (dataset, fname)	writes time series of data
<i>write_ts_grid_data</i> (dataset, fname)	writes time series of data
<i>write_ts_ml</i> (dt_ml, ml_top_avg, ml_top_std, ...)	writes time series of melting layer data
<i>write_ts_cum</i> (dataset, fname)	writes time series accumulation of data
<i>write_monitoring_ts</i> (start_time, np_t, ..., ...)	writes time series of data
<i>write_excess_gates</i> (excess_dict, fname)	Writes the position and values of gates that have a frequency of occurrence higher than a particular threshold
<i>write_intercomp_scores_ts</i> (start_time, stats, ...)	writes time series of radar intercomparison scores

Continued on next page

Table 1 – continued from previous page

<code>write_colocated_gates(coloc_gates, fname)</code>	Writes the position of gates colocated with two radars
<code>write_colocated_data(coloc_data, fname)</code>	Writes the data of gates colocated with two radars
<code>write_colocated_data_time_avg(coloc_data, fname)</code>	Writes the time averaged data of gates colocated with two radars
<code>write_sun_hits(sun_hits, fname)</code>	Writes sun hits data.
<code>write_sun_retrieval(sun_retrieval, fname)</code>	Writes sun retrieval data.

`pyrad.io.write_data.send_msg(sender, receiver_list, subject, fname)`  
sends the content of a text file by email

**Parameters**

**sender** [str] the email address of the sender  
**receiver\_list** [list of string] list with the email addresses of the receiver  
**subject** [str] the subject of the email  
**fname** [str] name of the file containing the content of the email message

**Returns**

**fname** [str] the name of the file containing the content

`pyrad.io.write_data.write_alarm_msg(radar_name, param_name_unit, date_last, target, tol_abs, np_trend, value_trend, tol_trend, nevents, np_last, value_last, fname)`

writes an alarm file

**Parameters**

**radar\_name** [str] Name of the radar being controlled  
**param\_name\_unit** [str] Parameter and units  
**date\_last** [datetime object] date of the current event  
**target, tol\_abs** [float] Target value and tolerance  
**np\_trend** [int] Total number of points in trend  
**value\_trend, tol\_trend** [float] Trend value and tolerance  
**nevents: int** Number of events in trend  
**np\_last** [int] Number of points in the current event  
**value\_last** [float] Value of the current event  
**fname** [str] Name of file where to store the alarm information

**Returns**

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_cdf(quantiles, values, ntot, nnan, nclut, nblocked, nprec_filter, noutliers, ncdf, fname, use_nans=False, nan_value=0.0, filterprec=[], vismin=None, sector=None, datatype=None, timeinfo=None)`

writes a cumulative distribution function

**Parameters**

**quantiles** [datetime array] array containing the measurement time  
**values** [float array] array containing the average value

**fname** [float array] array containing the standard deviation

**sector** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_colocated_data(coloc_data, fname)`

Writes the data of gates colocated with two radars

#### Parameters

**coloc\_data** [dict] dictionary containing the colocated data parameters

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_colocated_data_time_avg(coloc_data, fname)`

Writes the time averaged data of gates colocated with two radars

#### Parameters

**coloc\_data** [dict] dictionary containing the colocated data parameters

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_colocated_gates(coloc_gates, fname)`

Writes the position of gates colocated with two radars

#### Parameters

**coloc\_gates** [dict] dictionary containing the colocated gates parameters

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_excess_gates(excess_dict, fname)`

Writes the position and values of gates that have a frequency of occurrence higher than a particular threshold

#### Parameters

**excess\_dict** [dict] dictionary containing the gates parameters

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_field_coverage(quantiles, values, ele_start, ele_stop, azi_start,  
azi_stop, threshold, nvalid_min, datatype, time-  
info, fname)`

writes the quantiles of the coverage on a particular sector

#### Parameters

**quantiles** [datetime array] array containing the quantiles computed

**values** [float array] quantile value

**ele\_start, ele\_stop, azi\_start, azi\_stop** [float] The limits of the sector  
**threshold** [float] The minimum value to consider the data valid  
**nvalid\_min** [int] the minimum number of points to consider that there are values in a ray  
**datatype** [str] data type and units  
**timeinfo** [datetime object] the time stamp of the data  
**fname** [str] name of the file where to write the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_fixed_angle`(*time\_data*, *fixed\_angle*, *rad\_lat*, *rad\_lon*, *rad\_alt*,  
*fname*)

writes an output file with the fixed angle data

#### Parameters

**time\_data** [datetime object] The scan time  
**fixed\_angle** [float] The first fixed angle in the scan  
**rad\_lat, rad\_lon, rad\_alt** [float] Latitude, longitude [deg] and altitude [m MSL] of the radar  
**fname** [str] The name of the file where to write

#### Returns

**fname** [str] the name of the file containing the content

`pyrad.io.write_data.write_histogram`(*bin\_edges*, *values*, *fname*, *datatype*='undefined',  
*step*=0)

writes a histogram

#### Parameters

**bin\_edges** [float array] array containing the histogram bin edges  
**values** [int array] array containing the number of points in each bin  
**fname** [str] file name  
**datatype: str** The data type  
**step** [str] The bin step

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_intercomp_scores_ts`(*start\_time*, *stats*, *field\_name*,  
*fname*, *rad1\_name*='RADAR001',  
*rad2\_name*='RADAR002',  
*rewrite*=False)

writes time series of radar intercomparison scores

#### Parameters

**start\_time** [datetime object or array of date time objects] the time of the intercomparison  
**stats** [dict] dictionary containing the statistics  
**field\_name** [str] The name of the field  
**fname** [str] file name where to store the data

**rad1\_name, rad2\_name** [str] Name of the radars intercompared

**rewrite** [bool] if True a new file is created

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_last_state(datetime_last, fname)`

writes SwissMetNet data in format datetime, avg\_value, std\_value

#### Parameters

**datetime\_last** [datetime object] date and time of the last state

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_monitoring_ts(start_time, np_t, values, quantiles, datatype, fname, rewrite=False)`

writes time series of data

#### Parameters

**start\_time** [datetime object or array of date time objects] the time of the monitoring

**np\_t** [int or array of ints] the total number of points

**values: float array with 3 elements of array of arrays** the values at certain quantiles

**quantiles: float array with 3 elements** the quantiles computed

**datatype** [str] The data type

**fname** [str] file name where to store the data

**rewrite** [bool] if True a new file is created

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_proc_periods(start_times, end_times, fname)`

writes an output file containing start and stop times of periods to process

#### Parameters

**start\_times, end\_times** [datetime object] The starting and ending times of the periods

**fname** [str] The name of the file where to write

#### Returns

**fname** [str] the name of the file containing the content

`pyrad.io.write_data.write_quantiles(quantiles, values, fname, datatype='undefined')`

writes quantiles

#### Parameters

**quantiles** [float array] array containing the quantiles to write

**values** [float array] array containing the value of each quantile

**fname** [str] file name

**datatype: str** The data type

**Returns**

**fname** [str] the name of the file where data has written

```
pyrad.io.write_data.write_rhi_profile (hvec, data, nvalid_vec, labels, fname, datatype=None,  
                                         timeinfo=None, sector=None)
```

writes the values of an RHI profile in a text file

**Parameters**

**hvec** [float array] array containing the altitude in m MSL

**data** [list of float array] the quantities at each altitude

**nvalid\_vec** [int array] number of valid data points used to compute the quantiles

**labels** [list of strings] label specifying the quantities in data

**fname** [str] file name where to store the data

**datatype** [str] the data type

**timeinfo** [datetime object] time of the rhi profile

**sector** [dict] dictionary specyng the sector limits

**Returns**

**fname** [str] the name of the file where data has been written

```
pyrad.io.write_data.write_smn (datetime_vec, value_avg_vec, value_std_vec, fname)
```

writes SwissMetNet data in format datetime,avg\_value, std\_value

**Parameters**

**datetime\_vec** [datetime array] array containing the measurement time

**value\_avg\_vec** [float array] array containing the average value

**value\_std\_vec** [float array] array containing the standard deviation

**fname** [str] file name where to store the data

**Returns**

**fname** [str] the name of the file where data has written

```
pyrad.io.write_data.write_sun_hits (sun_hits, fname)
```

Writes sun hits data.

**Parameters**

**sun\_hits** [dict] dictionary containing the sun hits parameters

**fname** [str] file name where to store the data

**Returns**

**fname** [str] the name of the file where data has written

```
pyrad.io.write_data.write_sun_retrieval (sun_retrieval, fname)
```

Writes sun retrieval data.

**Parameters**

**sun\_retrieval** [dict] dictionary containing the sun retrieval parameters

**fname** [str] file name where to store the data

**Returns**



**fname** [str] the name of the file where data has written

```
pyrad.io.write_data.write_trt_cell_data(traj_ID, yyyyymmddHHMM, lon, lat, ell_L, ell_S,  
                                         ell_or, area, vel_x, vel_y, det, RANKr, CG_n,  
                                         CG_p, CG, CG_percent_p, ET45, ET45m, ET15,  
                                         ET15m, VIL, maxH, maxHm, POH, RANK, Dvel_x,  
                                         Dvel_y, cell_contour, fname)
```

writes TRT cell data

#### Parameters

**traj\_ID**, **yyyyymmddHHMM**, **lon**, **lat**, **ell\_L**, **ell\_S**, **ell\_or**, **area**,  
**vel\_x**, **vel\_y**, **det**, **RANKr**, **CG\_n**, **CG\_p**, **CG**, **CG\_percent\_p**, **ET45**,  
**ET45m**, **ET15**, **ET15m**, **VIL**, **maxH**, **maxHm**, **POH**, **RANK**, **Dvel\_x**,  
**Dvel\_y**, **cell\_contour**: the cell parameters  
**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

```
pyrad.io.write_data.write_trt_cell_lightning(cell_ID, cell_time, lon, lat, area, rank,  
                                              nflash, flash_density, fname, timefor-  
                                              mat='%Y%m%d%H%M')
```

writes the lightning data for each TRT cell

#### Parameters

**cell\_ID** [array of ints] the cell ID  
**cell\_time** [array of datetime] the time step  
**lon**, **lat** [array of floats] the latitude and longitude of the center of the cell  
**area** [array of floats] the area of the cell  
**rank** [array of floats] the rank of the cell  
**nflash** [array of ints] the number of flashes/sources within the cell  
**flash\_density** [array of floats] the flash/source density  
**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

```
pyrad.io.write_data.write_trt_cell_scores(traj_ID, flash_density_max_time,  
                                           flash_density_max_rank, nflashes_max_list,  
                                           area_flash_max_list, flash_density_max,  
                                           rank_max_time, rank_max, fname)
```

writes TRT cells scores

#### Parameters

**traj\_ID** [array of ints] The ID of the cells  
**flash\_density\_max\_time** [array of date times] The time at which the maximum flash density  
was reached for each cell  
**flash\_density\_max\_rank** [array of floats] The rank when the maximum flash density was  
reached for each cell

**nflashes\_max\_list** [array of ints] the number of flashes when the max flash density was reached

**area\_flash\_max\_list** [array of floats] The area when the max flash density was reached

**flash\_density\_max** [array of floats] The maximum flash density for each cell

**rank\_max\_time** [array of datetime] the time at wich the maximum rank of each cell was reached

**rank\_max** [array of float] the rank when the maximum rank of each cell was reached

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_trt_info(ids, max_rank, nscans, time_start, time_end, fname)`  
writes TRT info of the thundertracking

#### Parameters

**ids, max\_rank, nscans, time\_start, time\_end:** array the cell parameters

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_trt_rpc(cell_ID, cell_time, lon, lat, area, rank, hmin, hmax, freq, fname, timeformat='%Y%m%d%H%M')`  
writes the rimed particles column data for a TRT cell

#### Parameters

**cell\_ID** [array of ints] the cell ID

**cell\_time** [array of datetime] the time step

**lon, lat** [array of floats] the latitude and longitude of the center of the cell

**area** [array of floats] the area of the cell

**rank** [array of floats] the rank of the cell

**hmin, hmax** [array of floats] Minimum and maximum altitude of the rimed particle column

**freq** [array of floats] Frequency of the species constituting the rime particle column within the limits of it

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_trt_thundertracking_data(traj_ID, scan_ordered_time, scan_time, azi, rng, yyymmd-dHHMM, lon, lat, ell_L, ell_S, ell_or, area, vel_x, vel_y, det, RANKr, CG_n, CG_p, CG, CG_percent_p, ET45, ET45m, ET15, ET15m, VIL, maxH, maxHm, POH, RANK, Dvel_x, Dvel_y, cell_contour, fname)`

writes TRT cell data of the thundertracking scan

#### Parameters

**traj\_ID**, **scan\_ordered\_time**, **scan\_time**, **azi**, **rng**, **yyyymmddHHMM**, **lon**, **lat**,  
**ell\_L**, **ell\_S**, **ell\_or**, **area**, **vel\_x**, **vel\_y**, **det**, **RANKr**, **CG\_n**, **CG\_p**, **CG**,  
**CG\_percent\_p**, **ET45**, **ET45m**, **ET15**, **ET15m**, **VIL**, **maxH**, **maxHm**, **POH**, **RANK**,  
**Dvel\_x**, **Dvel\_y**, **cell\_contour**: the cell parameters  
**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_ts_cum(dataset, fname)`  
writes time series accumulation of data

#### Parameters

**dataset** [dict] dictionary containing the time series parameters  
**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_ts_grid_data(dataset, fname)`  
writes time series of data

#### Parameters

**dataset** [dict] dictionary containing the time series parameters  
**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

`pyrad.io.write_data.write_ts_lightning(flashnr, time_data, time_in_flash, lat, lon, alt, dBm,  
vals_list, fname, pol_vals_labels)`  
writes the LMA sources data and the value of the colocated polarimetric variables

#### Parameters

**flashnr** [int] flash number  
**time\_data** [datetime object] flash source time  
**time\_in\_flash** [float] seconds since start of flash  
**lat**, **lon**, **alt** [float] latitude, longitude [deg] and altitude [m MSL] of the flash source  
**dBm** [float] flash power  
**vals\_list** [list of arrays] List containing the data for each polarimetric variable  
**fname** [str] the name of the file containing the content  
**pol\_values\_labels** [list of strings] List containing strings identifying each polarimetric variable

#### Returns

**fname** [str] the name of the file containing the content

```
pyrad.io.write_data.write_ts_ml(dt_ml, ml_top_avg, ml_top_std, thick_avg, thick_std,  
                                nrays_valid, nrays_total, fname)
```

writes time series of melting layer data

#### Parameters

**dt\_ml** [date time array] array of time steps

**ml\_top\_avg, ml\_top\_std: float arrays** the average and the standard deviation of the melting layer top height

**thick\_avg, thick\_std: float arrays** the average and the standard deviation of the metling layer thickness

**nrays\_valid, nrays\_total: int arrays** the number of rays where melting layer has been identified and the total number of arrays in the scan

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

```
pyrad.io.write_data.write_ts_polar_data(dataset, fname)
```

writes time series of data

#### Parameters

**dataset** [dict] dictionary containing the time series parameters

**fname** [str] file name where to store the data

#### Returns

**fname** [str] the name of the file where data has written

## PYRAD.IO.TIMESERIES

TimeSeries class implementation for holding timeseries data.

---

<i>TimeSeries</i> (desc[, timevec, timeformat, ...])	Holding timeseries data and metadata.
--	---------------------------------------

---

```
class pyrad.io.timeseries.TimeSeries (desc,          timevec=None,          timeformat=None,
                                     maxlen=None, datatype="")
```

Bases: `object`

Holding timeseries data and metadata.

### Attributes

**description** [array of str] Description of the data of the time series.

**time\_vector** [array of datetime objects]

**timeformat** [how to print the time (default:)] 'Date, UTC [seconds since midnight]'

**dataseries** [List of \_dataSeries object holding the] data

### Methods

<i>add_dataseries</i> (self, label, unit_name, unit)	Add a new data series to the timeseries object.
<i>add_timesample</i> (self, dt, values)	Add a new sample to the time series.
<i>plot</i> (self, fname[, ymin, ymax])	Make a figure of a time series
<i>plot_hist</i> (self, fname[, step])	Make histograms of time series
<i>write</i> (self, fname)	Write time series output

---

**\_\_class\_\_**

alias of `builtins.type`

**\_\_delattr\_\_** (self, name, /)

Implement `delattr`(self, name).

**\_\_dict\_\_** = `mappingproxy`({'\_\_module\_\_': 'pyrad.io.timeseries', '\_\_doc\_\_': "\n Holding

**\_\_dir\_\_** (self, /)

Default `dir()` implementation.

**\_\_eq\_\_** (self, value, /)

Return `self==value`.

**\_\_format\_\_** (self, format\_spec, /)

Default object formatter.

**\_\_ge\_\_** (*self*, *value*, /)  
Return self>=value.

**\_\_getattr\_\_** (*self*, *name*, /)  
Return getattr(self, name).

**\_\_gt\_\_** (*self*, *value*, /)  
Return self>value.

**\_\_hash\_\_** (*self*, /)  
Return hash(self).

**\_\_init\_\_** (*self*, *desc*, *timevec*=None, *timeformat*=None, *maxlength*=None, *datatype*=")  
Initialize the object.

#### Parameters

**desc** [array of str]

**timevec** [array of datetime]

**timeformat** [specifies time format]

**maxlength** [Maximal length of the time series]

**num\_el** [Number of values in the time series]

**\_\_init\_subclass\_\_** ()  
This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

**\_\_le\_\_** (*self*, *value*, /)  
Return self<=value.

**\_\_lt\_\_** (*self*, *value*, /)  
Return self<value.

**\_\_module\_\_** = 'pyrad.io.timeseries'

**\_\_ne\_\_** (*self*, *value*, /)  
Return self!=value.

**\_\_new\_\_** (\*args, \*\*kwargs)  
Create and return a new object. See help(type) for accurate signature.

**\_\_reduce\_\_** (*self*, /)  
Helper for pickle.

**\_\_reduce\_ex\_\_** (*self*, *protocol*, /)  
Helper for pickle.

**\_\_repr\_\_** (*self*, /)  
Return repr(self).

**\_\_setattr\_\_** (*self*, *name*, *value*, /)  
Implement setattr(self, name, value).

**\_\_sizeof\_\_** (*self*, /)  
Size of object in memory, in bytes.

**\_\_str\_\_** (*self*, /)  
Return str(self).

**\_\_subclasshook\_\_** ()

Abstract classes can override this to customize issubclass().

This is invoked early on by `abc.ABCMeta.__subclasscheck__()`. It should return `True`, `False` or `NotImplemented`. If it returns `NotImplemented`, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**add\_dataserries** (*self*, *label*, *unit\_name*, *unit*, *dataseries=None*, *plot=True*, *color=None*, *linestyle=None*)

Add a new data series to the timeseries object. The length of the data vector must be the same as the length of the time vector.

**add\_timesample** (*self*, *dt*, *values*)

Add a new sample to the time series.

**plot** (*self*, *fname*, *ymin=None*, *ymax=None*)

Make a figure of a time series

**plot\_hist** (*self*, *fname*, *step=None*)

Make histograms of time series

**write** (*self*, *fname*)

Write time series output

**class** `pyrad.io.timeseries._DataSeries` (*label*, *unit\_name*, *unit*, *data*, *plot=True*, *color=None*, *linestyle=None*)

Bases: `object`

Hold a data vector and some meta information.

## Methods

*set\_value*(*self*, *i*, *val*)

Append value to array

---

**\_\_class\_\_**

alias of `builtins.type`

**\_\_delattr\_\_** (*self*, *name*, /)

Implement `delattr`(*self*, *name*).

**\_\_dict\_\_** = `mappingproxy({'__module__': 'pyrad.io.timeseries', '__doc__': '\n Hold a`

**\_\_dir\_\_** (*self*, /)

Default `dir()` implementation.

**\_\_eq\_\_** (*self*, *value*, /)

Return `self==value`.

**\_\_format\_\_** (*self*, *format\_spec*, /)

Default object formatter.

**\_\_ge\_\_** (*self*, *value*, /)

Return `self>=value`.

**\_\_getattr\_\_** (*self*, *name*, /)

Return `getattr`(*self*, *name*).

**\_\_gt\_\_** (*self, value, /*)  
Return self>value.

**\_\_hash\_\_** (*self, /*)  
Return hash(self).

**\_\_init\_\_** (*self, label, unit\_name, unit, data, plot=True, color=None, linestyle=None*)  
Initialize the object.

**\_\_init\_subclass\_\_** ()  
This method is called when a class is subclassed.  
  
The default implementation does nothing. It may be overridden to extend subclasses.

**\_\_le\_\_** (*self, value, /*)  
Return self<=value.

**\_\_lt\_\_** (*self, value, /*)  
Return self<value.

**\_\_module\_\_** = **'pyrad.io.timeseries'**

**\_\_ne\_\_** (*self, value, /*)  
Return self!=value.

**\_\_new\_\_** (\*args, \*\*kwargs)  
Create and return a new object. See help(type) for accurate signature.

**\_\_reduce\_\_** (*self, /*)  
Helper for pickle.

**\_\_reduce\_ex\_\_** (*self, protocol, /*)  
Helper for pickle.

**\_\_repr\_\_** (*self, /*)  
Return repr(self).

**\_\_setattr\_\_** (*self, name, value, /*)  
Implement setattr(self, name, value).

**\_\_sizeof\_\_** (*self, /*)  
Size of object in memory, in bytes.

**\_\_str\_\_** (*self, /*)  
Return str(self).

**\_\_subclasshook\_\_** ()  
Abstract classes can override this to customize issubclass().  
  
This is invoked early on by abc.ABCMeta.\_\_subclasscheck\_\_(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

**\_\_weakref\_\_**  
list of weak references to the object (if defined)

**set\_value** (*self, i, val*)  
Append value to array



## PYRAD.IO.TRAJECTORY

Trajectory class implementation for reading trajectory file. Converting to different coordinate systems.

<i>Trajectory</i> (filename[, starttime, endtime, ...])	A class for reading and handling trajectory data from a file.
<i>_Radar_Trajectory</i> (lat, lon, alt)	A class for holding the trajectory data assigned to a radar.

```
class pyrad.io.trajectory.Trajectory(filename,  starttime=None,  endtime=None,  traj-  
                                     type='plane', flashnr=0)
```

Bases: `object`

A class for reading and handling trajectory data from a file.

### Attributes

**filename** [str] Path and name of the trajectory definition file

**starttime** [datetime] Start time of trajectory processing.

**endtime** [datetime] End time of trajectory processing.

**trajtype** [str]

**Type of trajectory. Can be 'plane' or 'lightning'**

**time\_vector** [Array of datetime objects] Array containing the trajectory time samples

**wgs84\_lat\_deg** [Array of floats] WGS84 latitude samples in radian

**wgs84\_lon\_deg** [Array of floats] WGS84 longitude samples in radian

**wgs84\_alt\_m** [Array of floats] WGS84 altitude samples in m

**nsamples** [int]

**Number of samples in the trajectory**

**\_swiss\_grid\_done** [Bool] Indicates that conversion to Swiss coordinates has been performed

**swiss\_chy, swiss\_chx, swiss\_chh** [Array of floats] Swiss coordinates in m

**radar\_list** [list] List of radars for which trajectories are going to be computed

**flashnr** [int] For 'lightning' only. Number of flash for which trajectory data is going to be computed. If 0 all all flashes are going to be considered.

**time\_in\_flash** [array of floats] For 'lightning' only. Time within flash (sec)

**flashnr\_vec** [array of ints] For 'lightning' only. Flash number of each data sample

**dBm** [array of floats] For 'lightning' only. Lightning power (dBm)

## Methods

<code>add_radar(self, radar)</code>	Add the coordinates (WGS84 longitude, latitude and non WGS84 altitude) of a radar to the <code>radar_list</code> .
<code>calculate_velocities(self, radar)</code>	Calculate velocities.
<code>get_end_time(self)</code>	Get time of last trajectory sample.
<code>get_samples_in_period(self[, start, end])</code>	”
<code>get_start_time(self)</code>	Get time of first trajectory sample.

```
__class__
    alias of builtins.type

__delattr__ (self, name, /)
    Implement delattr(self, name).

__dict__ = mappingproxy({'__module__': 'pyrad.io.trajectory', '__doc__': "\n A class
__dir__ (self, /)
    Default dir() implementation.

__eq__ (self, value, /)
    Return self==value.

__format__ (self, format_spec, /)
    Default object formatter.

__ge__ (self, value, /)
    Return self>=value.

__getattr__ (self, name, /)
    Return getattr(self, name).

__gt__ (self, value, /)
    Return self>value.

__hash__ (self, /)
    Return hash(self).

__init__ (self, filename, starttime=None, endtime=None, trajtype='plane', flashnr=0)
    Initialize the object.
```

## Parameters

**filename** [str] Filename containing the trajectory samples.

**starttime** [datetime] Start time of trajectory processing. If not given, use the time of the first trajectory sample.

**endtime** [datetime] End time of trajectory processing. If not given, use the time of the last trajectory sample.

**trajtype** [str] type of trajectory. Can be plane or lightning

**flashnr** [int] If type of trajectory is lightning, the flash number to check the trajectory. 0 means all flash numbers included

```
__init_subclass__ ()
    This method is called when a class is subclassed.
```

The default implementation does nothing. It may be overridden to extend subclasses.

**\_\_le\_\_** (*self*, *value*, /)  
Return self<=value.

**\_\_lt\_\_** (*self*, *value*, /)  
Return self<value.

**\_\_module\_\_** = 'pyrad.io.trajectory'

**\_\_ne\_\_** (*self*, *value*, /)  
Return self!=value.

**\_\_new\_\_** (\**args*, \*\**kwargs*)  
Create and return a new object. See help(type) for accurate signature.

**\_\_reduce\_\_** (*self*, /)  
Helper for pickle.

**\_\_reduce\_ex\_\_** (*self*, *protocol*, /)  
Helper for pickle.

**\_\_repr\_\_** (*self*, /)  
Return repr(self).

**\_\_setattr\_\_** (*self*, *name*, *value*, /)  
Implement setattr(self, name, value).

**\_\_sizeof\_\_** (*self*, /)  
Size of object in memory, in bytes.

**\_\_str\_\_** (*self*, /)  
Return str(self).

**\_\_subclasshook\_\_** ()  
Abstract classes can override this to customize issubclass().  
  
This is invoked early on by abc.ABCMeta.\_\_subclasscheck\_\_(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

**\_\_weakref\_\_**  
list of weak references to the object (if defined)

**\_convert\_traj\_to\_swissgrid** (*self*)  
Convert trajectory samples from WGS84 to Swiss CH1903 coordinates

**\_get\_total\_seconds** (*self*, *x*)  
Return total seconds of timedelta object

**\_read\_traj** (*self*)  
Read trajectory from file

**\_read\_traj\_lightning** (*self*, *flashnr=0*)  
Read trajectory from lightning file

#### Parameters

**flashnr** [int] the flash number to keep. If 0 data from all flashes will be kept

**\_read\_traj\_trt** (*self*)  
Read trajectory from TRT file

**add\_radar** (*self*, *radar*)  
Add the coordinates (WGS84 longitude, latitude and non WGS84 altitude) of a radar to the radar\_list.

#### Parameters

**radar** [pyart radar object] containing the radar coordinates

**calculate\_velocities** (*self*, *radar*)  
Calculate velocities.

**get\_end\_time** (*self*)  
Get time of last trajectory sample.

**get\_samples\_in\_period** (*self*, *start=None*, *end=None*)  
” Get indices of samples of the trajectory within given time period.

**get\_start\_time** (*self*)  
Get time of first trajectory sample.

**class** pyrad.io.trajectory.\_Radar\_Trajectory (*lat*, *lon*, *alt*)  
Bases: `object`

A class for holding the trajectory data assigned to a radar.

#### Attributes

**latitude** [float] WGS84 radar latitude [deg]

**longitude** [float] WGS84 radar longitude [deg]

**altitude** [float] radar altitude [m] (non WGS84)

**ch\_y, ch\_x, ch\_alt** [float] radar coordinates in swiss CH1903 coordinates

**elevation\_vec** [float list] Elevation values of the trajectory samples

**azimuth\_vec** [float list] Azimuth values of the trajectory samples

**range\_vec** [float list] Range values of the trajectory samples

**v\_abs, v\_r, v\_el, v\_az** [array-like] Velocity vectors of the absolute [m/s], radial [m/s], elevation [deg/s] and azimuth [deg/s] velocities

#### Methods

<code>assign_trajectory(self, el, az, rr)</code>	Assign a trajectory to the radar in polar radar coordinates.
<code>assign_velocity_vecs(self, v_abs, v_r, v_el, ...)</code>	Assign velocity vectors to the radar.
<code>convert_radpos_to_swissgrid(self)</code>	Convert the radar location (in WGS84 coordinates) to swiss CH1903 coordinates.
<code>location_is_equal(self, lat, lon, alt)</code>	Check if the given coordinates are the same.

**\_\_class\_\_**  
alias of `builtins.type`

**\_\_delattr\_\_** (*self*, *name*, /)  
Implement `delattr(self, name)`.

**\_\_dict\_\_** = `mappingproxy({'__module__': 'pyrad.io.trajectory', '__doc__': '\n A class`

**\_\_dir\_\_** (*self*, /)  
Default `dir()` implementation.

**\_\_eq\_\_** (*self*, *value*, /)  
Return `self==value`.

**\_\_format\_\_** (*self, format\_spec, /*)  
Default object formatter.

**\_\_ge\_\_** (*self, value, /*)  
Return self>=value.

**\_\_getattr\_\_** (*self, name, /*)  
Return getattr(self, name).

**\_\_gt\_\_** (*self, value, /*)  
Return self>value.

**\_\_hash\_\_** (*self, /*)  
Return hash(self).

**\_\_init\_\_** (*self, lat, lon, alt*)  
Initialize the object.

#### Parameters

**lat, lon , alt** [radar location coordinates]

**nsamps** [number of samples]

**\_\_init\_subclass\_\_** ()  
This method is called when a class is subclassed.  
  
The default implementation does nothing. It may be overridden to extend subclasses.

**\_\_le\_\_** (*self, value, /*)  
Return self<=value.

**\_\_lt\_\_** (*self, value, /*)  
Return self<value.

**\_\_module\_\_** = 'pyrad.io.trajectory'

**\_\_ne\_\_** (*self, value, /*)  
Return self!=value.

**\_\_new\_\_** (\*args, \*\*kwargs)  
Create and return a new object. See help(type) for accurate signature.

**\_\_reduce\_\_** (*self, /*)  
Helper for pickle.

**\_\_reduce\_ex\_\_** (*self, protocol, /*)  
Helper for pickle.

**\_\_repr\_\_** (*self, /*)  
Return repr(self).

**\_\_setattr\_\_** (*self, name, value, /*)  
Implement setattr(self, name, value).

**\_\_sizeof\_\_** (*self, /*)  
Size of object in memory, in bytes.

**\_\_str\_\_** (*self, /*)  
Return str(self).

**\_\_subclasshook\_\_** ()  
Abstract classes can override this to customize issubclass().

This is invoked early on by `abc.ABCMeta.__subclasscheck__()`. It should return `True`, `False` or `NotImplemented`. If it returns `NotImplemented`, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

**`__weakref__`**

list of weak references to the object (if defined)

**`assign_trajectory`** (*self*, *el*, *az*, *rr*)

Assign a trajectory to the radar in polar radar coordinates.

**Parameters**

***el*, *az*, *rr*** [array-like] elevation, azimuth and range vector

**`assign_velocity_vecs`** (*self*, *v\_abs*, *v\_r*, *v\_el*, *v\_az*)

Assign velocity vectors to the radar.

**`convert_radpos_to_swissgrid`** (*self*)

Convert the radar location (in WGS84 coordinates) to swiss CH1903 coordinates.

**`location_is_equal`** (*self*, *lat*, *lon*, *alt*)

Check if the given coordinates are the same.

**Parameters**

***lat*, *lon*, *alt*** [radar location coordinates]

## PYRAD.GRAPH.PLOTS\_AUX

### Auxiliary plotting functions

<code>generate_complex_range_Doppler_title(radar, ...)</code>	creates the fixed range plot title
<code>generate_angle_Doppler_title(radar, field, ...)</code>	creates the angle-Doppler plot title
<code>generate_complex_Doppler_title(radar, field, ...)</code>	creates the fixed range plot title
<code>generate_fixed_rng_span_title(radar, field, stat)</code>	creates the fixed range plot title
<code>generate_fixed_rng_title(radar, field, fixed_rng)</code>	creates the fixed range plot title
<code>get_colobar_label(field_dict, field_name)</code>	creates the colorbar label using field metadata
<code>get_field_name(field_dict, field)</code>	Return a nice field name for a particular field
<code>get_norm(field_name)</code>	Computes the normalization of the colormap, and gets the ticks and labels of the colorbar from the metadata of the field.

`pyrad.graph.plots_aux.generate_angle_Doppler_title(radar, field, ang, ind_rng, along_azi=True, date-time_format=None)`

creates the angle-Doppler plot title

#### Parameters

**radar** [radar] The radar object

**field** [str] name of the field

**ang** [float] The fixed angle

**ind\_rng** [int] the index of the fixed range

**along\_azi** [bool] If true the plot is performed along azimuth, otherwise it is performed along elevation

**datetime\_forat** [str or None] The date time format to use

#### Returns

**titl** [str] The plot title

`pyrad.graph.plots_aux.generate_complex_Doppler_title(radar, field, ray, rng, date-time_format=None)`

creates the fixed range plot title

**Parameters**

**radar** [radar] The radar object  
**field** [str] name of the field  
**stat** [str] The statistic computed  
**datetime\_forat** [str or None] The date time format to use

**Returns**

**titl** [str] The plot title

`pyrad.graph.plots_aux.generate_complex_range_Doppler_title(radar, field, ray, date-  
time_format=None)`  
creates the fixed range plot title

**Parameters**

**radar** [radar] The radar object  
**field** [str] name of the field  
**stat** [str] The statistic computed  
**datetime\_forat** [str or None] The date time format to use

**Returns**

**titl** [str] The plot title

`pyrad.graph.plots_aux.generate_fixed_rng_span_title(radar, field, stat, date-  
time_format=None)`  
creates the fixed range plot title

**Parameters**

**radar** [radar] The radar object  
**field** [str] name of the field  
**stat** [str] The statistic computed  
**datetime\_forat** [str or None] The date time format to use

**Returns**

**titl** [str] The plot title

`pyrad.graph.plots_aux.generate_fixed_rng_title(radar, field, fixed_rng, date-  
time_format=None)`  
creates the fixed range plot title

**Parameters**

**radar** [radar] The radar object  
**field** [str] name of the field  
**fixed\_rng** [float] The fixed range [m]  
**datetime\_forat** [str or None] The date time format to use

**Returns**

**titl** [str] The plot title

`pyrad.graph.plots_aux.get_colobar_label(field_dict, field_name)`  
creates the colorbar label using field metadata



**Parameters**

**field\_dict** [dict] dictionary containing field metadata

**field\_name** [str] name of the field

**Returns**

**label** [str] colorbar label

`pyrad.graph.plots_aux.get_field_name(field_dict, field)`

Return a nice field name for a particular field

**Parameters**

**field\_dict** [dict] dictionary containing field metadata

**field** [str] name of the field

**Returns**

**field\_name** [str] the field name

`pyrad.graph.plots_aux.get_norm(field_name)`

Computes the normalization of the colormap, and gets the ticks and labels of the colorbar from the metadata of the field. Returns None if the required parameters are not present in the metadata

**Parameters**

**field\_name** [str] name of the field

**Returns**

**norm** [list] the colormap index

**ticks** [list] the list of ticks in the colorbar

**labels** [list] the list of labels corresponding to each tick



## PYRAD.GRAPH.PLOTS

Functions to plot Pyrad datasets

<i>plot_pos</i> (lat, lon, alt, fname_list[, ax, ...])	plots a trajectory on a Cartesian surface
<i>plot_pos_map</i> (lat, lon, alt, fname_list[, ...])	plots a trajectory on a map
<i>plot_density</i> (hist_obj, hist_type, ...[, ...])	density plot (angle-values representation)
<i>plot_scatter</i> (bin_edges1, bin_edges2, ...[, ...])	2D histogram
<i>plot_quantiles</i> (quant, value, fname_list[, ...])	plots quantiles
<i>plot_histogram</i> (bin_edges, values, fname_list)	computes and plots histogram
<i>plot_histogram2</i> (bin_centers, hist, fname_list)	plots histogram
<i>plot_antenna_pattern</i> (antpattern, fname_list)	plots an antenna pattern
<i>plot_selfconsistency</i> (zdrkdp_table, fname_list)	plots a ZDR-KDP/ZH selfconsistency in rain relation
<i>plot_selfconsistency_instrument</i> (zdr, kdp, zh, ...)	plots the ZDR-KDP/ZH relationship obtained by an instrument.
<i>plot_scatter_comp</i> (value1, value2, fname_list)	plots the scatter between two time series
<i>plot_sun_hits</i> (field, field_name, fname_list, ...)	plots the sun hits
<i>_plot_time_range</i> (rad_time, rad_range, ...[, ...])	plots a time-range plot

`pyrad.graph.plots._plot_time_range`(*rad\_time*, *rad\_range*, *rad\_data*, *field\_name*, *fname\_list*, *titl*='Time-Range plot', *xlabel*='time (s from start time)', *ylabel*='range (Km)', *clabel*=None, *vmin*=None, *vmax*=None, *figsize*=[10, 8], *save\_fig*=True, *dpi*=72)

plots a time-range plot

### Parameters

**rad\_time** [1D array] array containing the x dimension (typically time)

**rad\_range** [1D array] array containing the y dimension (typically range)

**rad\_data** [2D array] array containing the data to plot

**field\_name** [str or None] field name. Used to define plot characteristics

**fname\_list** [list of str] list of names of the files where to store the plot

**titl** [str] Plot title

**xlabel, ylabel** [str] x- and y-axis labels

**clabel** [str or None] colorbar label

**vmin, vmax** [float] min and max values of the color bar

**figsize** [list] figure size [xsize, ysize]

**save\_fig** [bool] If true the figure is saved in files `fname_list`. Otherwise the fig and ax is returned

**dpi** [int] dpi

#### Returns

**fname\_list** [list of str] list of names of the created plots or

**fig, ax** [object] handles to fig and ax objects

`pyrad.graph.plots.plot_antenna_pattern` (*antpattern, fname\_list, labelx='Angle [Deg]', linear=False, twoway=False, title='Antenna Pattern', ymin=None, ymax=None, dpi=72*)

plots an antenna pattern

#### Parameters

**antpattern** [dict] dictionary with the angle and the attenuation

**value** [float array] values of the time series

**fname\_list** [list of str] list of names of the files where to store the plot

**labelx** [str] The label of the X axis

**linear** [boolean] if true data is in linear units

**linear** [boolean] if true data represents the two way attenuation

**titl** [str] The figure title

**ymin, ymax: float** Lower/Upper limit of y axis

**dpi** [int] dots per inch

#### Returns

**fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots.plot_density` (*hist\_obj, hist\_type, field\_name, ind\_sweep, prdcfg, fname\_list, quantiles=[25.0, 50.0, 75.0], ref\_value=0.0, vmin=None, vmax=None*)

density plot (angle-values representation)

#### Parameters

**hist\_obj** [histogram object] object containing the histogram data to plot

**hist\_type** [str] type of histogram (instantaneous data or cumulative)

**field\_name** [str] name of the radar field to plot

**ind\_sweep** [int] sweep index to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**quantiles** [array] the quantile lines to plot

**ref\_value** [float] the reference value

**vmin, vmax** [float] Minim and maximum extend of the vertical axis

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots.plot_histogram(bin_edges, values, fname_list, labelx='bins', la-  
bely='Number of Samples', titl='histogram', dpi=72)
```

computes and plots histogram

#### Parameters

**bin\_edges** [array] histogram bin edges

**values** [array] data values

**fname\_list** [list of str] list of names of the files where to store the plot

**labelx** [str] The label of the X axis

**labeledy** [str] The label of the Y axis

**titl** [str] The figure title

**dpi** [int] dots per inch

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots.plot_histogram2(bin_centers, hist, fname_list, width=None, labelx='bins',  
labeledy='Number of Samples', titl='histogram', dpi=72,  
ax=None, fig=None, save_fig=True, color=None, al-  
pha=None, invert_xaxis=False)
```

plots histogram

#### Parameters

**bin\_centers** [array] histogram bin centers

**hist** [array] values for each bin

**fname\_list** [list of str] list of names of the files where to store the plot

**width** [scalar or array-like] the width(s) of the bars. If None it is going to be estimated from the distances between centers

**labelx** [str] The label of the X axis

**labeledy** [str] The label of the Y axis

**titl** [str] The figure title

**dpi** [int] dots per inch

**fig** [Figure] Figure to add the colorbar to. If none a new figure will be created

**ax** [Axis] Axis to plot on. if fig is None a new axis will be created

**save\_fig** [bool] if true save the figure. If false it does not close the plot and returns the handle to the figure

**color** [str] color of the bars

**alpha** [float] parameter controlling the transparency

**invert\_xaxis** [bool] If true inverts the x axis

#### Returns

**fname\_list or fig, ax: list of str** list of names of the created plots

```
pyrad.graph.plots.plot_pos(lat, lon, alt, fname_list, ax=None, fig=None, save_fig=True,
                           sort_altitude='No', dpi=72, alpha=1.0, cb_label='height [m MSL]',
                           titl='Position', xlabel='Lon [Deg]', ylabel='Lat [Deg]', limits=None,
                           vmin=None, vmax=None)
```

plots a trajectory on a Cartesian surface

#### Parameters

**lat, lon, alt** [float array] Points coordinates

**fname\_list** [list of str] list of names of the files where to store the plot

**fig** [Figure] Figure to add the colorbar to. If none a new figure will be created

**ax** [Axis] Axis to plot on. if fig is None a new axis will be created

**save\_fig** [bool] if true save the figure if false it does not close the plot and returns the handle to the figure

**sort\_altitude** [str] String indicating whether to sort the altitude data. Can be 'No', 'Lowest\_on\_top' or 'Highest\_on\_top'

**dpi** [int] Pixel density

**alpha** [float] Transparency

**cb\_label** [str] Color bar label

**titl** [str] Plot title

**xlabel, ylabel** [str] The labels of the X and Y axis

**limits** [tuple or None] The limits of the field to plot

**vmin, vmax** [float] The limits of the color scale

#### Returns

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure and axes

```
pyrad.graph.plots.plot_pos_map(lat, lon, alt, fname_list, ax=None, fig=None, save_fig=True,
                               sort_altitude='No', dpi=72, alpha=1.0, cb_label='height
                               [m MSL]', titl='Position', xlabel='Lon [Deg]', ylabel='Lat
                               [Deg]', limits=None, vmin=None, vmax=None, lon_step=0.3,
                               lat_step=0.1, background_zoom=8)
```

plots a trajectory on a map

#### Parameters

**lat, lon, alt** [float array] Points coordinates

**fname\_list** [list of str] list of names of the files where to store the plot

**fig** [Figure] Figure to add the colorbar to. If none a new figure will be created

**ax** [Axis] Axis to plot on. if fig is None a new axis will be created

**save\_fig** [bool] if true save the figure if false it does not close the plot and returns the handle to the figure

**sort\_altitude** [str] String indicating whether to sort the altitude data. Can be 'No', 'Lowest\_on\_top' or 'Highest\_on\_top'

**dpi** [int] Pixel density

**alpha** [float] Transparency

**cb\_label** [str] Color bar label

**titl** [str] Plot title

**xlabel, ylabel** [str] The labels of the X and Y axis

**limits** [tuple or None] The limits of the field to plot

**vmin, vmax** [float] The limits of the color scale

**lon\_step, lat\_step** [float] The step interval of the latitude, longitude lines to plot

**background\_zoom** [int] The zoom of the background image. A higher number will give more level of detail at the expense of speed.

#### Returns

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes

`pyrad.graph.plots.plot_quantiles` (*quant, value, fname\_list, labelx='quantile', labely='value', titl='quantile', vmin=None, vmax=None, dpi=72*)

plots quantiles

#### Parameters

**quant** [array] quantiles to be plotted

**value** [array] values of each quantile

**fname\_list** [list of str] list of names of the files where to store the plot

**labelx** [str] The label of the X axis

**labely** [str] The label of the Y axis

**titl** [str] The figure title

**vmin, vmax: float** Lower/Upper limit of data values

**dpi** [int] dots per inch

#### Returns

**fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots.plot_scatter` (*bin\_edges1, bin\_edges2, hist\_2d, field\_name1, field\_name2, fname\_list, prdcfg, metadata=None, lin\_regr=None, lin\_regr\_slope1=None, rad1\_name='RADAR001', rad2\_name='RADAR002'*)

2D histogram

#### Parameters

**bin\_edges1, bin\_edges2** [float array2] the bins of each field

**hist\_2d** [ndarray 2D] the 2D histogram

**field\_name1, field\_name2** [str] the names of each field

**fname\_list** [list of str] list of names of the files where to store the plot

**prdcfg** [dict] product configuration dictionary

**metadata** [str] a string with metadata to write in the plot

**lin\_regr** [tuple with 2 values] the coefficients for a linear regression

**lin\_regr\_slope1** [float] the intercept point of a linear regression of slope 1

**rad1\_name, rad2\_name** [str] name of the radars which data is used

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots.plot_scatter_comp(value1, value2, fname_list, labelx='Sensor 1',  
                                     labely='Sensor 2', titl='Scatter', axis=None,  
                                     metadata=None, dpi=72, ax=None, fig=None,  
                                     save_fig=True, point_format='bx')
```

plots the scatter between two time series

#### Parameters

**value1** [float array] values of the first time series

**value2** [float array] values of the second time series

**fname\_list** [list of str] list of names of the files where to store the plot

**labelx** [str] The label of the X axis

**labely** [str] The label of the Y axis

**titl** [str] The figure title

**axis** [str] type of axis

**metadata** [string] a string containing metadata

**dpi** [int] dots per inch

**fig** [Figure] Figure to add the colorbar to. If none a new figure will be created

**ax** [Axis] Axis to plot on. if fig is None a new axis will be created

**save\_fig** [bool] if true save the figure if false it does not close the plot and returns the handle to the figure

**point\_format** [str] format of the scatter point

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots.plot_selfconsistency(zdrkdp_table, fname_list, labelx='ZDR [dB]',  
                                        labely='KDP/Zh [(deg*m3)/(km*mm6)]', ti-  
                                        tle='Selfconsistency in rain', ymin=None,  
                                        ymax=None, dpi=72, save_fig=True, ax=None,  
                                        fig=None)
```

plots a ZDR-KDP/ZH selfconsistency in rain relation

#### Parameters

**antpattern** [dict] dictionary with the angle and the attenuation

**value** [float array] values of the time series

**fname\_list** [list of str] list of names of the files where to store the plot

**labelx** [str] The label of the X axis

**linear** [boolean] if true data is in linear units

**linear** [boolean] if true data represents the two way attenuation

**titl** [str] The figure title

**ymin, ymax: float** Lower/Upper limit of y axis



**dpi** [int] dots per inch

### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots.plot_selfconsistency_instrument (zdr, kdp, zh, fname_list,
                                                    bins_zdr_step=0.05,
                                                    bins_zdr_min=0.0,
                                                    bins_zdr_max=6.0,
                                                    bins_kdpzh_step=0.1,
                                                    bins_kdpzh_min=-2.0,
                                                    bins_kdpzh_max=20.0, normalize=True,
                                                    vmin=0.0, vmax=0.01, parametrization='None',
                                                    zdr_kdpzh_dict=None, retrieve_relation=True,
                                                    plot_theoretical=True, dpi=72)
```

plots the ZDR-KDP/ZH relationship obtained by an instrument. The theoretical curve and the retrieved curve

### Parameters

**zdr, kdp, zh** [1D ndarray] The valid values of ZDR, KDP and Zh collected by the instrument

**fname\_list** [list of str] list of names of the files where to store the plot

**bins\_zdr\_step** [float] The step of the ZDR axis of the histogram [dB]

**bins\_zdr\_min, bins\_zdr\_max** [float] The limits of the ZDR axis of the histogram (bins center) [dB]

**bins\_kdpzh\_step** [float] The step of the  $1e5 * KDP^a / ZH^b$  axis of the histogram [(deg\*m3)/(km\*mm6)]

**bins\_kdpzh\_min, bins\_kdpzh\_max** [float] The limits of the  $1e5 * KDP^a / ZH^b$  axis of the histogram (bins center) [(deg\*m3)/(km\*mm6)]

**normalize** [Bool] If True the occurrence density of ZK/KDP for each ZDR bin is going to be represented. Otherwise it will show the number of gates at each bin

**vmin, vmax** [float] min and max values of the colorbar

**parametrization** [str] The type of parametrization for the self-consistency curves. Can be 'None', 'Gourley', 'Wolfensberger', 'Louf', 'Gorgucci' or 'Vaccarono'. 'None' will use tables contained in zdr\_kdpzh\_dict. The parametrized curves are obtained from literature except for Wolfensberger that was derived from disdrometer data obtained by MeteoSwiss and EPFL. All parametrizations are valid for C-band only except that of Gourley.

**zdr\_kdpzh\_dict** [dict] dictionary containing a look up table relating ZDR with KDP/Zh for different elevations and the frequency band of the radar

**retrieve\_relation** [boolean] if true a zdr-kdp/zh relationship is retrieved from the data

**plot\_theoretical** [bool] if true the theoretical relationship is retrieved

**dpi** [int] dots per inch

### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots.plot_sun_hits (field, field_name, fname_list, prdcfg)
plots the sun hits
```

#### Parameters

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**altitude** [float] the altitude [m MSL] to be plotted

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

#### Returns

**fname\_list** [list of str] list of names of the created plots

## PYRAD.GRAPH.PLOTS\_VOL

Functions to plot radar volume data

<i>plot_ray</i> (radar, field_name, ind_ray, prdcfg, ...)	plots a ray
<i>plot_ppi</i> (radar, field_name, ind_el, prdcfg, ...)	plots a PPI
<i>plot_ppi_map</i> (radar, field_name, ind_el, ...)	plots a PPI on a geographic map
<i>plot_rhi</i> (radar, field_name, ind_az, prdcfg, ...)	plots an RHI
<i>plot_bscope</i> (radar, field_name, ind_sweep, ...)	plots a B-Scope (angle-range representation)
<i>plot_time_range</i> (radar, field_name, ...)	plots a time-range plot
<i>plot_fixed_rng</i> (radar, field_name, prdcfg, ...)	plots a fixed range plot
<i>plot_fixed_rng_span</i> (radar, field_name, ...)	plots a fixed range plot
<i>plot_cappi</i> (radar, field_name, altitude, ...)	plots a Constant Altitude Plan Position Indicator CAPPI
<i>plot_traj</i> (rng_traj, azi_traj, ele_traj, ...)	plots a trajectory on a Cartesian surface
<i>plot_rhi_contour</i> (radar, field_name, ind_az, ...)	plots contour data on an RHI
<i>plot_ppi_contour</i> (radar, field_name, ind_el, ...)	plots contour data on a PPI
<i>plot_roi_contour</i> (roi_dict, prdcfg, fname_list)	plots the contour of a region of interest on a map
<i>plot_rhi_profile</i> (data_list, hvec, fname_list)	plots an RHI profile
<i>plot_along_coord</i> (xval_list, yval_list, ...)	plots data along a certain radar coordinate
<i>plot_field_coverage</i> (xval_list, yval_list, ...)	plots a time series

`pyrad.graph.plots_vol.plot_along_coord(xval_list, yval_list, fname_list, labelx='coord', labely='Value', labels=None, title='Plot along coordinate', colors=None, linestyle=None, ymin=None, ymax=None, dpi=72)`

plots data along a certain radar coordinate

### Parameters

**xval\_list** [list of float arrays] the x values, range, azimuth or elevation

**yval\_list** [list of float arrays] the y values. Parameter to plot

**fname\_list** [list of str] list of names of the files where to store the plot

**labelx** [str] The label of the X axis

**labely** [str] The label of the Y axis

**labels** [array of str] The label of the legend

**title** [str] The figure title

**colors** [array of str] Specifies the colors of each line

**linestyles** [array of str] Specifies the line style of each line

**ymin, ymax: float** Lower/Upper limit of y axis

**dpi** [int] dots per inch

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots_vol.plot_bscope(radar, field_name, ind_sweep, prdcfg, fname_list,  
                                   vmin=None, vmax=None, ray_dim='ang',  
                                   xaxis_rng=True)
```

plots a B-Scope (angle-range representation)

#### Parameters

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**ind\_sweep** [int] sweep index to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**vmin, vmax** [Min and max values of the colorbar]

**ray\_dim** [str] the ray dimension. Can be 'ang' or 'time'

**xaxis** [bool] if true the range will be in the x-axis. Otherwise it will be in the y-axis.

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots_vol.plot_cappi(radar, field_name, altitude, prdcfg, fname_list,  
                                   beamwidth=1.0, beam_spacing=1.0, save_fig=True)
```

plots a Constant Altitude Plan Position Indicator CAPPI

#### Parameters

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**altitude** [float] the altitude [m MSL] to be plotted

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**beamwidth** [float] The radar beamwidth

**beam\_spacing** [float] the ray angle resolution

**save\_fig** [bool] if true save the figure. If false it does not close the plot and returns the handle to the figure

#### Returns

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes

```
pyrad.graph.plots_vol.plot_field_coverage(xval_list, yval_list, fname_list, labelx='Azimuth  
(deg)', labely='Range extension [m]', la-  
bels=None, title='Field coverage', ymin=None,  
ymax=None, xmeanval=None, ymeanval=None,  
labelmeanval=None, dpi=72)
```

plots a time series

**Parameters**

**xval\_list** [list of float arrays] the x values, azimuth  
**yval\_list** [list of float arrays] the y values. Range extension  
**fname\_list** [list of str] list of names of the files where to store the plot  
**labelx** [str] The label of the X axis  
**labely** [str] The label of the Y axis  
**labels** [array of str] The label of the legend  
**title** [str] The figure title  
**ymin, ymax** [float] Lower/Upper limit of y axis  
**xmeanval, ymeanval** [float array] the x and y values of a mean along elevation  
**labelmeanval** [str] the label of the mean  
**dpi** [int] dots per inch

**Returns**

**fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots_vol.plot_fixed_rng(radar, field_name, prdcfg, fname_list, azi_res=None, ele_res=None, ang_tol=1.0, vmin=None, vmax=None)`  
plots a fixed range plot

**Parameters**

**radar** [radar object] The radar object containing the fixed range data  
**field\_name** [str] The name of the field to plot  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot  
**azi\_res, ele\_res** [float] The nominal azimuth and elevation angle resolution [deg]  
**ang\_tol** [float] The tolerance between the nominal and the actual radar angle  
**vmin, vmax** [float] Min and Max values of the color scale. If None it is going to be taken from the Py-ART config files

**Returns**

**fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots_vol.plot_fixed_rng_span(radar, field_name, prdcfg, fname_list, azi_res=None, ele_res=None, ang_tol=1.0, stat='max')`  
plots a fixed range plot

**Parameters**

**radar** [radar object] The radar object containing the fixed range data  
**field\_name** [str] The name of the field to plot  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot  
**azi\_res, ele\_res** [float] The nominal azimuth and elevation angle resolution [deg]

**ang\_tol** [float] The tolerance between the nominal and the actual radar angle

#### Returns

**fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots_vol.plot_ppi(radar, field_name, ind_el, prdcfg, fname_list, plot_type='PPI',  
titl=None, vmin=None, vmax=None, step=None, quan-  
tiles=None, save_fig=True)`

plots a PPI

#### Parameters

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**ind\_el** [int] sweep index to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**plot\_type** [str] type of plot (PPI, QUANTILES or HISTOGRAM)

**titl** [str] Plot title

**vmin, vmax** [float] The minimum and maximum value. If None the scale is going to be obtained from the Py-ART config file.

**step** [float] step for histogram plotting

**quantiles** [float array] quantiles to plot

**save\_fig** [bool] if true save the figure. If false it does not close the plot and returns the handle to the figure

#### Returns

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes

`pyrad.graph.plots_vol.plot_ppi_contour(radar, field_name, ind_el, prdcfg, fname_list,  
contour_values=None, linewidths=1.5, ax=None,  
fig=None, save_fig=True)`

plots contour data on a PPI

#### Parameters

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**ind\_el** [int] sweep index to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**contour\_values** [float array] list of contours to plot

**linewidths** [float] width of the contour lines

**fig** [Figure] Figure to add the colorbar to. If none a new figure will be created

**ax** [Axis] Axis to plot on. if fig is None a new axis will be created

**save\_fig** [bool] if true save the figure if false it does not close the plot and returns the handle to the figure

#### Returns

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes

```
pyrad.graph.plots_vol.plot_ppi_map(radar, field_name, ind_el, prdcfg, fname_list,  
                                     save_fig=True)
```

plots a PPI on a geographic map

#### Parameters

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**ind\_el** [int] sweep index to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**save\_fig** [bool] if true save the figure. If false it does not close the plot and returns the handle to the figure

#### Returns

**fname\_list** [list of str or]

**fig, ax, display** [tuple] list of names of the saved plots or handle of the figure an axes

```
pyrad.graph.plots_vol.plot_ray(radar, field_name, ind_ray, prdcfg, fname_list, titl=None,  
                                vmin=None, vmax=None, save_fig=True)
```

plots a ray

#### Parameters

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**ind\_ray** [int] ray index to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**plot\_type** [str] type of plot (PPI, QUANTILES or HISTOGRAM)

**titl** [str] Plot title

**vmin, vmax** [float] min and max values of the y axis

**save\_fig** [bool] if true save the figure. If false it does not close the plot and returns the handle to the figure

#### Returns

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes

```
pyrad.graph.plots_vol.plot_rhi(radar, field_name, ind_az, prdcfg, fname_list, plot_type='RHI',  
                                titl=None, vmin=None, vmax=None, step=None, quan-  
                                tiles=None, save_fig=True)
```

plots an RHI

**Parameters**

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**ind\_az** [int] sweep index to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**plot\_type** [str] type of plot (PPI, QUANTILES or HISTOGRAM)

**titl** [str] Plot title

**vmin, vmax** [float] The minimum and maximum value. If None the scale is going to be obtained from the Py-ART config file.

**step** [float] step for histogram plotting

**quantiles** [float array] quantiles to plot

**save\_fig** [bool] if true save the figure. If false it does not close the plot and returns the handle to the figure

**Returns**

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes

`pyrad.graph.plots_vol.plot_rhi_contour(radar, field_name, ind_az, prdcfg, fname_list, contour_values=None, linewidths=1.5, ax=None, fig=None, save_fig=True)`

plots contour data on an RHI

**Parameters**

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**ind\_az** [int] sweep index to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**contour\_values** [float array] list of contours to plot

**linewidths** [float] width of the contour lines

**fig** [Figure] Figure to add the colorbar to. If none a new figure will be created

**ax** [Axis] Axis to plot on. if fig is None a new axis will be created

**save\_fig** [bool] if true save the figure if false it does not close the plot and returns the handle to the figure

**Returns**

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes



`pyrad.graph.plots_vol.plot_rhi_profile` (*data\_list, hvec, fname\_list, labelx='Value', labely='Height (m MSL)', labels=['Mean'], title='RHI profile', colors=None, linestyle=None, vmin=None, vmax=None, hmin=None, hmax=None, dpi=72*)

plots an RHI profile

#### Parameters

**data\_list** [list of float array] values of the profile  
**hvec** [float array] height points of the profile  
**fname\_list** [list of str] list of names of the files where to store the plot  
**labelx** [str] The label of the X axis  
**labely** [str] The label of the Y axis  
**labels** [array of str] The label of the legend  
**title** [str] The figure title  
**colors** [array of str] Specifies the colors of each line  
**linestyle** [array of str] Specifies the line style of each line  
**vmin, vmax: float** Lower/Upper limit of data values  
**hmin, hmax: float** Lower/Upper limit of altitude  
**dpi** [int] dots per inch

#### Returns

**fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots_vol.plot_roi_contour` (*roi\_dict, prdcfg, fname\_list, plot\_center=True, xlabel='Lon [Deg]', ylabel='Lat [Deg]', titl='TRT cell position', ax=None, fig=None, save\_fig=True*)

plots the contour of a region of interest on a map

#### Parameters

**roi\_dict** [dict] dictionary containing lon\_roi, lat\_roi, the points defining the contour  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot  
**plot\_center** [bool] If True a marked with the center of the roi is plotted  
**fig** [Figure] Figure to add the colorbar to. If none a new figure will be created  
**ax** [Axis] Axis to plot on. if fig is None a new axis will be created  
**save\_fig** [bool] if true save the figure if false it does not close the plot and returns the handle to the figure

#### Returns

**fname\_list** [list of str or]  
**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes

`pyrad.graph.plots_vol.plot_time_range` (*radar, field\_name, ind\_sweep, prdcfg, fname\_list*)  
plots a time-range plot

#### Parameters

**radar** [Radar object] object containing the radar data to plot

**field\_name** [str] name of the radar field to plot

**ind\_sweep** [int] sweep index to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

#### Returns

**fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots_vol.plot_traj(rng_traj, azi_traj, ele_traj, time_traj, prdcfg, fname_list, rad_alt=None, rad_tstart=None, ax=None, fig=None, save_fig=True)`

plots a trajectory on a Cartesian surface

#### Parameters

**rng\_traj, azi\_traj, ele\_traj** [float array] antenna coordinates of the trajectory [m and deg]

**time\_traj** [datetime array] trajectory time

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**rad\_alt** [float or None] radar altitude [m MSL]

**rad\_tstart** [datetime object or None] start time of the radar scan

**surface\_alt** [float] surface altitude [m MSL]

**color\_ref** [str] What the color code represents. Can be 'None', 'rel\_altitude', 'altitude' or 'time'

**fig** [Figure] Figure to add the colorbar to. If none a new figure will be created

**ax** [Axis] Axis to plot on. if fig is None a new axis will be created

**save\_fig** [bool] if true save the figure if false it does not close the plot and returns the handle to the figure

#### Returns

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes

## PYRAD.GRAPH.PLOTS\_GRID

Functions to plot data in a Cartesian grid format

<i>plot_surface</i> (grid, field_name, level, ...[, ...])	plots a surface from gridded data
<i>plot_surface_contour</i> (grid, field_name, ...)	plots a surface from gridded data
<i>plot_latitude_slice</i> (grid, field_name, lon, ...)	plots a latitude slice from gridded data
<i>plot_longitude_slice</i> (grid, field_name, lon, ...)	plots a longitude slice from gridded data
<i>plot_latlon_slice</i> (grid, field_name, coord1, ...)	plots a croos section crossing two points in the grid

`pyrad.graph.plots_grid.plot_latitude_slice` (*grid*, *field\_name*, *lon*, *lat*, *prdcfg*, *fname\_list*)  
plots a latitude slice from gridded data

### Parameters

- grid** [Grid object] object containing the gridded data to plot
- field\_name** [str] name of the radar field to plot
- lon, lat** [float] coordinates of the slice to plot
- prdcfg** [dict] dictionary containing the product configuration
- fname\_list** [list of str] list of names of the files where to store the plot

### Returns

- fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots_grid.plot_latlon_slice` (*grid*, *field\_name*, *coord1*, *coord2*, *prdcfg*, *fname\_list*)  
plots a croos section crossing two points in the grid

### Parameters

- grid** [Grid object] object containing the gridded data to plot
- field\_name** [str] name of the radar field to plot
- coord1** [tuple of floats] lat, lon of the first point
- coord2** [tuple of floats] lat, lon of the second point
- fname\_list** [list of str] list of names of the files where to store the plot

### Returns

- fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots_grid.plot_longitude_slice` (*grid, field\_name, lon, lat, prdcfg, fname\_list*)  
plots a longitude slice from gridded data

**Parameters**

**grid** [Grid object] object containing the gridded data to plot  
**field\_name** [str] name of the radar field to plot  
**lon, lat** [float] coordinates of the slice to plot  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot

**Returns**

**fname\_list** [list of str] list of names of the created plots

`pyrad.graph.plots_grid.plot_surface` (*grid, field\_name, level, prdcfg, fname\_list, titl=None, save\_fig=True, use\_basemap=False*)  
plots a surface from gridded data

**Parameters**

**grid** [Grid object] object containing the gridded data to plot  
**field\_name** [str] name of the radar field to plot  
**level** [int] level index  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot  
**titl** [str] Plot title  
**save\_fig** [bool] if true save the figure. If false it does not close the plot and returns the handle to the figure

**Returns**

**fname\_list** [list of str or]  
**fig, ax, display** [tuple] list of names of the saved plots or handle of the figure an axes

`pyrad.graph.plots_grid.plot_surface_contour` (*grid, field\_name, level, prdcfg, fname\_list, contour\_values=None, linewidths=1.5, ax=None, fig=None, display=None, save\_fig=True, use\_basemap=False*)  
plots a surface from gridded data

**Parameters**

**grid** [Grid object] object containing the gridded data to plot  
**field\_name** [str] name of the radar field to plot  
**level** [int] level index  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot  
**contour\_values** [float array] list of contours to plot  
**linewidths** [float] width of the contour lines  
**fig** [Figure] Figure to add the colorbar to. If none a new figure will be created

**ax** [Axis] Axis to plot on. if fig is None a new axis will be created

**save\_fig** [bool] if true save the figure if false it does not close the plot and returns the handle to the figure

**Returns**

**fname\_list** [list of str or]

**fig, ax** [tuple] list of names of the saved plots or handle of the figure an axes



## PYRAD.GRAPH.PLOTS\_SPECTRA

Functions to plot spectral data

<i>plot_range_Doppler</i> (spectra, field_name, ray, ...)	Makes a range-Doppler plot
<i>plot_angle_Doppler</i> (spectra, field_name, ang, ...)	Makes an angle-Doppler plot
<i>plot_time_Doppler</i> (spectra, field_name, ...)	Makes a time-Doppler plot
<i>plot_Doppler</i> (spectra, field_name, ray, rng, ...)	Makes a Doppler plot
<i>plot_complex_range_Doppler</i> (spectra, ...[, ...])	Makes a complex range-Doppler plot.
<i>plot_complex_angle_Doppler</i> (spectra, ...[, ...])	Makes an angle-Doppler plot of complex spectra
<i>plot_complex_time_Doppler</i> (spectra, ...[, ...])	Makes a complex time-Doppler plot.
<i>plot_amp_phase_range_Doppler</i> (spectra, ...[, ...])	Makes a complex range-Doppler plot plotting separately the module and the phase of the signal
<i>plot_amp_phase_angle_Doppler</i> (spectra, ...[, ...])	Makes an angle-Doppler plot of complex spectra
<i>plot_amp_phase_time_Doppler</i> (spectra, ...[, ...])	Makes a complex time-Doppler plot plotting separately the module and the phase of the signal
<i>plot_complex_Doppler</i> (spectra, field_name, ...)	Makes a complex Doppler plot plotting separately the real and the imaginary parts
<i>plot_amp_phase_Doppler</i> (spectra, field_name, ...)	Makes a complex Doppler plot plotting separately the module and the phase of the signal
<i>_create_irregular_grid</i> (xaxis, yaxis[, yaxis_pos])	Create an irregular grid to be able to plot data with variable x-axis
<i>_adapt_data_to_irregular_grid</i> (data, ...)	Adapts data to irregular grid to allow plotting

`pyrad.graph.plots_spectra._adapt_data_to_irregular_grid`(data, *axis\_size*,  
*nxbin\_lim*, *nybin\_lim*)

Adapts data to irregular grid to allow plotting

### Parameters

- data** [2D float array] The data to plot
- axis\_size** [1D-array] The size of each x-axis
- nxbin\_lim, nybin\_lim** [float] number of gate limits at each axis

### Returns

- zpoints** [2D float array] Matrix containing the data points.

`pyrad.graph.plots_spectra._create_irregular_grid` (*xaxis*, *yaxis*, *yaxis\_pos*=*'start'*)

Create an irregular grid to be able to plot data with variable x-axis

#### Parameters

**xaxis** [2D float array] array containing the x points

**yaxis** [1D float array] array containing the y points

**yaxis\_pos** [str] the position that the y point represents in the y-axis bin. Can be *'start'*, *'end'* or *'centre'*

#### Returns

**xaxis\_lim**, **yaxis\_lim** [2D float array] Matrix containing the edges of the X and Y axis

`pyrad.graph.plots_spectra.plot_Doppler` (*spectra*, *field\_name*, *ray*, *rng*, *prdcfg*, *fname\_list*,  
*xaxis\_info*=*'Doppler\_velocity'*, *ylabel*=*None*,  
*titl*=*None*, *vmin*=*None*, *vmax*=*None*)

Makes a Doppler plot

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot

**field\_name** [str] name of the field to plot

**ray**, **rng** [int] ray and rng index

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**xaxis\_info** [str] Type of x-axis. Can be *'Doppler\_velocity'*, *'Doppler\_frequency'* or *'pulse\_number'*

**ylabel** [str or None] The label of the y-axis

**titl** [str or None] The plot title

**vmin**, **vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

`pyrad.graph.plots_spectra.plot_amp_phase_Doppler` (*spectra*, *field\_name*, *ray*,  
*rng*, *prdcfg*, *fname\_list*,  
*xaxis\_info*=*'Doppler\_velocity'*,  
*titl*=*None*, *ampli\_vmin*=*None*, *ampli\_vmax*=*None*, *phase\_vmin*=*None*,  
*phase\_vmax*=*None*)

Makes a complex Doppler plot plotting separately the module and the phase of the signal

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot

**field\_name** [str] name of the field to plot

**ray**, **rng** [int] ray and range index

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**xaxis\_info** [str] Type of x-axis. Can be *'Doppler\_velocity'*, *'Doppler\_frequency'* or *'pulse\_number'*



**titl** [str or None] The plot title

**ampli\_vmin, ampli\_vmax, phase\_vmin, phase\_vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

```
pyrad.graph.plots_spectra.plot_amp_phase_angle_Doppler(spectra,          field_name,
                                                         ang,   ind_rays,  ind_rng,
                                                         prdcfg,         fname_list,
                                                         xaxis_info='Doppler_velocity',
                                                         yaxis_pos='centre',
                                                         along_azi=True, titl=None,
                                                         ampli_vmin=None,
                                                         ampli_vmax=None,
                                                         phase_vmin=None,
                                                         phase_vmax=None)
```

Makes an angle-Doppler plot of complex spectra

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot

**field\_name** [str] name of the field to plot

**ang** [float] The fixed angle

**ind\_rays** [1D int array] The indices of the rays to plot

**ind\_rng** [int] The index of the range to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**yaxis\_pos** [str] the position that the y point represents in the y-axis bin. Can be 'start', 'end' or 'centre'

**along\_azi** [bool] If true the plot is performed along azimuth. If false it is performed along elevation

**titl** [str or None] The plot title

**ampli\_vmin, ampli\_vmax, phase\_vmin, phase\_vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

```
pyrad.graph.plots_spectra.plot_amp_phase_range_Doppler(spectra,          field_name,
                                                         ray,   prdcfg,  fname_list,
                                                         xaxis_info='Doppler_velocity',
                                                         titl=None,          am-
                                                         pli_vmin=None,      am-
                                                         pli_vmax=None,
                                                         phase_vmin=None,
                                                         phase_vmax=None)
```

Makes a complex range-Doppler plot plotting separately the module and the phase of the signal

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot

**field\_name** [str] name of the field to plot

**ray** [int] ray index

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**titl** [str or None] The plot title

**ampli\_vmin, ampli\_vmax, phase\_vmin, phase\_vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

`pyrad.graph.plots_spectra.plot_amp_phase_time_Doppler` (*spectra*, *field\_name*,  
*prdcfg*, *fname\_list*,  
*xaxis\_info*='Doppler\_velocity',  
*yaxis\_pos*='start', *titl*=None,  
*ampli\_vmin*=None,  
*ampli\_vmax*=None,  
*phase\_vmin*=None,  
*phase\_vmax*=None)

Makes a complex time-Doppler plot plotting separately the module and the phase of the signal

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot

**field\_name** [str] name of the field to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**yaxis\_pos** [str] the position that the y point represents in the y-axis bin. Can be 'start', 'end' or 'centre'

**titl** [str or None] The plot title

**ampli\_vmin, ampli\_vmax, phase\_vmin, phase\_vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

`pyrad.graph.plots_spectra.plot_angle_Doppler` (*spectra*, *field\_name*, *ang*, *ind\_rays*,  
*ind\_rng*, *prdcfg*, *fname\_list*,  
*xaxis\_info*='Doppler\_velocity',  
*yaxis\_pos*='centre', *along\_azi*=True,  
*titl*=None, *clabel*=None, *vmin*=None,  
*vmax*=None)

Makes an angle-Doppler plot

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot

**field\_name** [str] name of the field to plot  
**ang** [float] The fixed angle  
**ind\_rays** [1D int array] The indices of the rays to plot  
**ind\_rng** [int] The index of the range to plot  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot  
**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'  
**yaxis\_pos** [str] the position that the y point represents in the y-axis bin. Can be 'start', 'end' or 'centre'  
**along\_azi** [bool] If true the plot is performed along azimuth. If false it is performed along elevation  
**titl** [str or None] The plot title  
**clabel** [str or None] The color bar label  
**vmin, vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

`pyrad.graph.plots_spectra.plot_complex_Doppler` (*spectra*, *field\_name*, *ray*,  
*rng*, *prdcfg*, *fname\_list*,  
*xaxis\_info*='Doppler\_velocity', *ylab*=*None*, *titl*=*None*, *vmin*=*None*,  
*vmax*=*None*)

Makes a complex Doppler plot plotting separately the real and the imaginary parts

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot  
**field\_name** [str] name of the field to plot  
**ray, rng** [int] ray and range index  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot  
**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'  
**ylabel** [str or None] The label of the y-axis  
**titl** [str or None] The plot title  
**vmin, vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

```
pyrad.graph.plots_spectra.plot_complex_angle_Doppler(spectra,          field_name,
                                                         ang,      ind_rays,    ind_rng,
                                                         prdcfg,      fname_list,
                                                         xaxis_info='Doppler_velocity',
                                                         yaxis_pos='centre',
                                                         along_azi=True,    titl=None,
                                                         clabel=None,      vmin=None,
                                                         vmax=None)
```

Makes an angle-Doppler plot of complex spectra

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot

**field\_name** [str] name of the field to plot

**ang** [float] The fixed angle

**ind\_rays** [1D int array] The indices of the rays to plot

**ind\_rng** [int] The index of the range to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**yaxis\_pos** [str] the position that the y point represents in the y-axis bin. Can be 'start', 'end' or 'centre'

**along\_azi** [bool] If true the plot is performed along azimuth. If false it is performed along elevation

**titl** [str or None] The plot title

**clabel** [str or None] The color bar label

**vmin, vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

```
pyrad.graph.plots_spectra.plot_complex_range_Doppler(spectra,          field_name,
                                                         ray,      prdcfg,    fname_list,
                                                         xaxis_info='Doppler_velocity',
                                                         titl=None,    clabel=None,
                                                         vmin=None, vmax=None)
```

Makes a complex range-Doppler plot. Plotting separately the real and the imaginary part

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot

**field\_name** [str] name of the field to plot

**ray** [int] ray index

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**titl** [str or None] The plot title  
**clabel** [str or None] The label of color bar  
**vmin, vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

```
pyrad.graph.plots_spectra.plot_complex_time_Doppler(spectra, field_name,  
                                                    prdcfg, fname_list,  
                                                    xaxis_info='Doppler_velocity',  
                                                    yaxis_pos='start', titl=None,  
                                                    clabel=None, vmin=None,  
                                                    vmax=None)
```

Makes a complex time-Doppler plot. Plotting separately the real and the imaginary part

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot  
**field\_name** [str] name of the field to plot  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot  
**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'  
**yaxis\_pos** [str] the position that the y point represents in the y-axis bin. Can be 'start', 'end' or 'centre'  
**titl** [str or None] The plot title  
**clabel** [str or None] The label of color bar  
**vmin, vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

```
pyrad.graph.plots_spectra.plot_range_Doppler(spectra, field_name, ray, prdcfg, fname_list,  
                                              xaxis_info='Doppler_velocity', titl=None,  
                                              clabel=None, vmin=None, vmax=None)
```

Makes a range-Doppler plot

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot  
**field\_name** [str] name of the field to plot  
**ray** [int] ray index  
**prdcfg** [dict] dictionary containing the product configuration  
**fname\_list** [list of str] list of names of the files where to store the plot  
**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'  
**titl** [str or None] The plot title  
**clabel** [str or None] The color bar label

**vmin, vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

```
pyrad.graph.plots_spectra.plot_time_Doppler(spectra, field_name, prdcfg, fname_list,  
                                             xaxis_info='Doppler_velocity',  
                                             yaxis_pos='start', titl=None, clabel=None,  
                                             vmin=None, vmax=None)
```

Makes a time-Doppler plot

#### Parameters

**spectra** [radar spectra object] object containing the spectra or the IQ data to plot

**field\_name** [str] name of the field to plot

**prdcfg** [dict] dictionary containing the product configuration

**fname\_list** [list of str] list of names of the files where to store the plot

**xaxis\_info** [str] Type of x-axis. Can be 'Doppler\_velocity', 'Doppler\_frequency' or 'pulse\_number'

**yaxis\_pos** [str] the position that the y point represents in the y-axis bin. Can be 'start', 'end' or 'centre'

**titl** [str or None] The plot title

**clabel** [str or None] The color bar label

**vmin, vmax** [float or None] The value limits

#### Returns

**fname\_list** [list of str] list of names of the saved plots

## PYRAD.GRAPH.PLOT\_TIMESERIES

Functions to plot Pyrad datasets

<i>plot_timeseries</i> (tvec, data_list, fname_list)	plots a time series
<i>plot_timeseries_comp</i> (date1, value1, date2, ...)	plots 2 time series in the same graph
<i>plot_monitoring_ts</i> (date, np_t, cquant, ...)	plots a time series of monitoring data
<i>plot_intercomp_scores_ts</i> (date_vec, np_vec, ...)	plots a time series of radar intercomparison scores
<i>plot_ml_ts</i> (dt_ml_arr, ml_top_avg_arr, ..., ...)	plots a time series of melting layer data
<i>plot_sun_retrieval_ts</i> (sun_retrieval, ..., ...)	plots sun retrieval time series series

```
pyrad.graph.plots_timeseries.plot_intercomp_scores_ts(date_vec, np_vec, mean-  
bias_vec, median-  
bias_vec, quant25bias_vec,  
quant75bias_vec, mode-  
bias_vec, corr_vec,  
slope_vec, intercep_vec,  
intercep_slope1_vec,  
fname_list, ref_value=0.0,  
np_min=0, corr_min=0.0,  
labelx='Time UTC',  
titl='RADAR001-RADAR002  
intercomparison', dpi=72)
```

plots a time series of radar intercomparison scores

### Parameters

**date\_vec** [datetime object] time of the time series

**np\_vec** [int array] number of points

**meanbias\_vec, medianbias\_vec, modebias\_vec** [float array] mean, median and mode bias

**quant25bias\_vec, quant75bias\_vec:** 25th and 75th percentile of the bias

**corr\_vec** [float array] correlation

**slope\_vec, intercep\_vec** [float array] slope and intercep of a linear regression

**intercep\_slope1\_vec** [float] the intercep point of a inear regression of slope 1

**ref\_value** [float] the reference value

**np\_min** [int] The minimum number of points to consider the result valid

**corr\_min** [float] The minimum correlation to consider the results valid

**labelx** [str] The label of the X axis

**titl** [str] The figure title

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots_timeseries.plot_ml_ts(dt_ml_arr, ml_top_avg_arr, ml_top_std_arr,
                                         thick_avg_arr, thick_std_arr, nrays_valid_arr,
                                         nrays_total_arr, fname_list, labelx='Time UTC',
                                         titl='Melting layer time series', dpi=72)
```

plots a time series of melting layer data

#### Parameters

**dt\_ml\_arr** [datetime object] time of the time series

**np\_vec** [int array] number of points

**meanbias\_vec, medianbias\_vec, modebias\_vec** [float array] mean, median and mode bias

**quant25bias\_vec, quant75bias\_vec:** 25th and 75th percentile of the bias

**corr\_vec** [float array] correlation

**slope\_vec, intercep\_vec** [float array] slope and intercep of a linear regression

**intercep\_slope1\_vec** [float] the intercep point of a inear regression of slope 1

**ref\_value** [float] the reference value

**np\_min** [int] The minimum number of points to consider the result valid

**corr\_min** [float] The minimum correlation to consider the results valid

**labelx** [str] The label of the X axis

**titl** [str] The figure title

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots_timeseries.plot_monitoring_ts(date, np_t, cquant, lquant,
                                                hquant, field_name, fname_list,
                                                ref_value=None, vmin=None,
                                                vmax=None, np_min=0, labelx='Time
[UTC]', labely='Value', titl='Time
Series', dpi=72)
```

plots a time series of monitoring data

#### Parameters

**date** [datetime object] time of the time series

**np\_t** [int array] number of points

**cquant, lquant, hquant** [float array] values of the central, low and high quantiles

**field\_name** [str] name of the field

**fname\_list** [list of str] list of names of the files where to store the plot

**ref\_value** [float] the reference value

**vmin, vmax** [float] The limits of the y axis

**np\_min** [int] minimum number of points to consider the sample plotable



**labelx** [str] The label of the X axis

**labeledy** [str] The label of the Y axis

**titl** [str] The figure title

**dpi** [int] dots per inch

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots_timeseries.plot_sun_retrieval_ts(sun_retrieval, data_type,  
                                                    fname_list, labelx='Date',  
                                                    titl='Sun retrieval Time Series',  
                                                    dpi=72)
```

plots sun retrieval time series series

#### Parameters

**sun\_retrieval** [tuple] tuple containing the retrieved parameters

**data\_type** [str] parameter to be plotted

**fname\_list** [list of str] list of names of the files where to store the plot

**labelx** [str] the x label

**titl** [str] the title of the plot

**dpi** [int] dots per inch

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots_timeseries.plot_timeseries(tvec, data_list, fname_list, labelx='Time  
[UTC]', labeledy='Value', labels=['Sensor'],  
                                              title='Time Series', period=0, timeformat  
                                              mat=None, colors=None, linestyles=None,  
                                              markers=None, ymin=None, ymax=None,  
                                              dpi=72)
```

plots a time series

#### Parameters

**tvec** [datetime object] time of the time series

**data\_list** [list of float array] values of the time series

**fname\_list** [list of str] list of names of the files where to store the plot

**labelx** [str] The label of the X axis

**labeledy** [str] The label of the Y axis

**labels** [array of str] The label of the legend

**title** [str] The figure title

**period** [float] measurement period in seconds used to compute accumulation. If 0 no accumulation is computed

**timeformat** [str] Specifies the tvec and time format on the x axis

**colors** [array of str] Specifies the colors of each line

**linestyles** [array of str] Specifies the line style of each line

**markers:** array of str Specify the markers to be used for each line

**ymin, ymax:** float Lower/Upper limit of y axis

**dpi** [int] dots per inch

#### Returns

**fname\_list** [list of str] list of names of the created plots

```
pyrad.graph.plots_timeseries.plot_timeseries_comp(date1, value1, date2, value2,
                                                    fname_list, labelx='Time [UTC]',
                                                    labely='Value', label1='Sensor
                                                    1', label2='Sensor 2', titl='Time
                                                    Series Comparison', period1=0, pe-
                                                    riod2=0, ymin=None, ymax=None,
                                                    dpi=72)
```

plots 2 time series in the same graph

#### Parameters

**date1** [datetime object] time of the first time series

**value1** [float array] values of the first time series

**date2** [datetime object] time of the second time series

**value2** [float array] values of the second time series

**fname\_list** [list of str] list of names of the files where to store the plot

**labelx** [str] The label of the X axis

**labely** [str] The label of the Y axis

**label1, label2** [str] legend label for each time series

**titl** [str]

The figure title

**period1, period2** [float] measurement period in seconds used to compute accumulation.  
If 0 no accumulation is computed

**dpi** [int] dots per inch

**ymin, ymax** [float] The limits of the Y-axis. None will keep the default limit.

#### Returns

**fname\_list** [list of str] list of names of the created plots

---

## PYRAD.UTIL.RADAR\_UTILS

Miscellaneous functions dealing with radar data

<i>get_data_along_rng</i> (radar, field_name, ..., ...)	Get data at particular (azimuths, elevations)
<i>get_data_along_az</i> (radar, field_name, ..., ...)	Get data at particular (ranges, elevations)
<i>get_data_along_ele</i> (radar, field_name, ..., ...)	Get data at particular (ranges, azimuths)
<i>get_ROI</i> (radar, fieldname, sector)	filter out any data outside the region of interest defined by sector
<i>rainfall_accumulation</i> (t_in_vec, val_in_vec)	Computes the rainfall accumulation of a time series over a given period
<i>time_series_statistics</i> (t_in_vec, val_in_vec)	Computes statistics over a time-averaged series.
<i>find_contiguous_times</i> (times[, step])	Given and array of ordered times, find those contiguous according to a maximum time step
<i>join_time_series</i> (t1, val1, t2, val2[, dropnan])	joins time_series.
<i>get_range_bins_to_avg</i> (rad1_rng, rad2_rng)	Compares the resolution of two radars and determines if and which radar has to be averaged and the length of the averaging window
<i>belongs_roi_indices</i> (lat, lon, roi)	Get the indices of points that belong to roi in a list of points
<i>find_ray_index</i> (ele_vec, azi_vec, ele, azi[, ...])	Find the ray index corresponding to a particular elevation and azimuth
<i>find_rng_index</i> (rng_vec, rng[, rng_tol])	Find the range index corresponding to a particular range
<i>find_nearest_gate</i> (radar, lat, lon[, latlon_tol])	Find the radar gate closest to a lat,lon point
<i>find_neighbour_gates</i> (radar, azi, rng[, ...])	Find the neighbouring gates within +-delta_azi and +-delta_rng
<i>find_colocated_indexes</i> (radar1, radar2, ...)	Given the theoretical elevation, azimuth and range of the co-located gates of two radars and a given tolerance returns the indices of the gates for the current radars
<i>get_target_elevations</i> (radar_in)	Gets RHI target elevations
<i>time_avg_range</i> (timeinfo, avg_starttime, ...)	finds the new start and end time of an averaging
<i>get_closest_solar_flux</i> (hit_datetime_list, ...)	finds the solar flux measurement closest to the sun hit
<i>get_fixed_rng_data</i> (radar, field_names, fixed_rng)	Creates a 2D-grid with (azi, ele) data at a fixed range
<i>create_sun_hits_field</i> (rad_el, rad_az, ...)	creates a sun hits field from the position and power of the sun hits

Continued on next page

Table 1 – continued from previous page

<code>create_sun_retrieval_field(par, field_name, ...)</code>	creates a sun retrieval field from the retrieval parameters
<code>compute_quantiles(field[, quantiles])</code>	computes quantiles
<code>compute_quantiles_from_hist(bin_centers, hist)</code>	computes quantiles from histograms
<code>compute_quantiles_sweep(field, ray_start, ...)</code>	computes quantiles of a particular sweep
<code>compute_histogram(field, field_name[, ...])</code>	computes histogram of the data
<code>compute_histogram_sweep(field, ray_start, ...)</code>	computes histogram of the data in a particular sweep
<code>get_histogram_bins(field_name[, step])</code>	gets the histogram bins using the range limits of the field as defined in the Py-ART config file.
<code>compute_2d_stats(field1, field2, ...[, ...])</code>	computes a 2D histogram and statistics of the data
<code>compute_1d_stats(field1, field2)</code>	returns statistics of data
<code>compute_2d_hist(field1, field2, field_name1, ...)</code>	computes a 2D histogram of the data
<code>quantize_field(field, field_name, step)</code>	quantizes data
<code>compute_profile_stats(field, gate_altitude, ...)</code>	Compute statistics of vertical profile
<code>compute_directional_stats(field[, avg_type, ...])</code>	Computes the mean or the median along one of the axis (ray or range)
<code>project_to_vertical(data_in, data_height, ...)</code>	Projects radar data to a regular vertical grid

`pyrad.util.radar_utils.belongs_roi_indices` (*lat, lon, roi*)

Get the indices of points that belong to roi in a list of points

#### Parameters

**lat, lon** [float arrays] latitudes and longitudes to check

**roi** [dict] Dictionary describing the region of interest

#### Returns

**inds** [array of ints] list of indices of points belonging to ROI

**is\_roi** [str] Whether the list of points is within the region of interest. Can be 'All', 'None', 'Some'

`pyrad.util.radar_utils.compute_1d_stats` (*field1, field2*)

returns statistics of data

#### Parameters

**field1, field2** [ndarray 1D] the two fields to compare

#### Returns

**stats** [dict] a dictionary with statistics

`pyrad.util.radar_utils.compute_2d_hist` (*field1, field2, field\_name1, field\_name2, step1=None, step2=None*)

computes a 2D histogram of the data

#### Parameters

**field1, field2** [ndarray 2D] the radar fields

**field\_name1, field\_name2** [str] field names

**step1, step2** [float] size of the bins

#### Returns

**H** [float array 2D] The bi-dimensional histogram of samples x and y

**xedges, yedges** [float array] the bin edges along each dimension

`pyrad.util.radar_utils.compute_2d_stats` (*field1, field2, field\_name1, field\_name2, step1=None, step2=None*)

computes a 2D histogram and statistics of the data

#### Parameters

**field1, field2** [ndarray 2D] the two fields

**field\_name1, field\_name2:** **str** the name of the fields

**step1, step2** [float] size of bin

#### Returns

**hist\_2d** [array] the histogram

**bin\_edges1, bin\_edges2** [float array] The bin edges

**stats** [dict] a dictionary with statistics

`pyrad.util.radar_utils.compute_directional_stats` (*field, avg\_type='mean', nvalid\_min=1, axis=0*)

Computes the mean or the median along one of the axis (ray or range)

#### Parameters

**field** [ndarray] the radar field

**avg\_type:** **str** the type of average: 'mean' or 'median'

**nvalid\_min** [int] the minimum number of points to consider the stats valid. Default 1

**axis** [int] the axis along which to compute (0=ray, 1=range)

#### Returns

**values** [ndarray 1D] The resultant statistics

**nvalid** [ndarray 1D] The number of valid points used in the computation

`pyrad.util.radar_utils.compute_histogram` (*field, field\_name, bin\_edges=None, step=None, vmin=None, vmax=None*)

computes histogram of the data

#### Parameters

**field** [ndarray 2D] the radar field

**field\_name:** **str or none** name of the field

**bin\_edges:** **ndarray 1D** the bin edges

**step** [float] size of bin

**vmin, vmax** [float] The minimum and maximum value of the histogram

#### Returns

**bin\_edges** [float array] interval of each bin

**values** [float array] values at each bin

`pyrad.util.radar_utils.compute_histogram_sweep` (*field, ray\_start, ray\_end, field\_name, step=None*)

computes histogram of the data in a particular sweep

**Parameters**

**field** [ndarray 2D] the radar field  
**ray\_start, ray\_end** [int] starting and ending ray indexes  
**field\_name: str** name of the field  
**step** [float] size of bin

**Returns**

**bin\_edges** [float array] interval of each bin  
**values** [float array] values at each bin

`pyrad.util.radar_utils.compute_profile_stats` (*field, gate\_altitude, h\_vec, h\_res, quantity='quantiles', quantiles=array([0.25, 0.5, 0.75]), nvalid\_min=4, std\_field=None, np\_field=None, make\_linear=False, include\_nans=False*)

Compute statistics of vertical profile

**Parameters**

**field** [ndarray] the radar field  
**gate\_altitude: ndarray** the altitude at each radar gate [m MSL]  
**h\_vec** [1D ndarray] height vector [m MSL]  
**h\_res** [float] height resolution [m]  
**quantity** [str] The quantity to compute. Can be ['quantiles', 'mode', 'regression\_mean', 'mean']. If 'mean', the min, max, and average is computed.  
**quantiles** [1D ndarray] the quantiles to compute  
**nvalid\_min** [int] the minimum number of points to consider the stats valid  
**std\_field** [ndarray] the standard deviation of the regression at each range gate  
**np\_field** [ndarray] the number of points used to compute the regression at each range gate  
**make\_linear** [Boolean] If true the data is transformed into linear coordinates before taking the mean  
**include\_nans** [Boolean] If true NaN will be considered as zeros

**Returns**

**vals** [ndarray 2D] The resultant statistics  
**val\_valid** [ndarray 1D] The number of points to compute the stats used at each height level

`pyrad.util.radar_utils.compute_quantiles` (*field, quantiles=None*)  
computes quantiles

**Parameters**

**field** [ndarray 2D] the radar field  
**ray\_start, ray\_end** [int] starting and ending ray indexes  
**quantiles: float array** list of quantiles to compute

**Returns**

**quantiles** [float array] list of quantiles

**values** [float array] values at each quantile

`pyrad.util.radar_utils.compute_quantiles_from_hist` (*bin\_centers*, *hist*, *quantiles=None*)

computes quantiles from histograms

#### Parameters

**bin\_centers** [ndarray 1D] the bins

**hist** [ndarray 1D] the histogram

**quantiles: float array** list of quantiles to compute

#### Returns

**quantiles** [float array] list of quantiles

**values** [float array] values at each quantile

`pyrad.util.radar_utils.compute_quantiles_sweep` (*field*, *ray\_start*, *ray\_end*, *quantiles=None*)

computes quantiles of a particular sweep

#### Parameters

**field** [ndarray 2D] the radar field

**ray\_start, ray\_end** [int] starting and ending ray indexes

**quantiles: float array** list of quantiles to compute

#### Returns

**quantiles** [float array] list of quantiles

**values** [float array] values at each quantile

`pyrad.util.radar_utils.create_sun_hits_field` (*rad\_el*, *rad\_az*, *sun\_el*, *sun\_az*, *data*, *imgcfg*)

creates a sun hits field from the position and power of the sun hits

#### Parameters

**rad\_el, rad\_az, sun\_el, sun\_az** [ndarray 1D] azimuth and elevation of the radar and the sun respectively in degree

**data** [masked ndarray 1D] the sun hit data

**imgcfg: dict** a dictionary specifying the ranges and resolution of the field to create

#### Returns

**field** [masked ndarray 2D] the sun hit field

`pyrad.util.radar_utils.create_sun_retrieval_field` (*par*, *field\_name*, *imgcfg*, *lant=0.0*)

creates a sun retrieval field from the retrieval parameters

#### Parameters

**par** [ndarray 1D] the 5 retrieval parameters

**imgcfg: dict** a dictionary specifying the ranges and resolution of the field to create

#### Returns

**field** [masked ndarray 2D] the sun retrieval field

`pyrad.util.radar_utils.find_ang_index` (*ang\_vec*, *ang*, *ang\_tol=0.0*)

Find the angle index corresponding to a particular fixed angle

**Parameters**

**ang\_vec** [float array] The angle data array where to look for

**ang** [float] The angle to search

**ang\_tol** [float] Tolerance [deg]

**Returns**

**ind\_ang** [int] The angle index

`pyrad.util.radar_utils.find_colocated_indexes(radar1, radar2, rad1_ele, rad1_azimuth, rad1_rng, rad2_ele, rad2_azimuth, rad2_rng, ele_tol=0.5, azi_tol=0.5, rng_tol=50.0)`

Given the theoretical elevation, azimuth and range of the co-located gates of two radars and a given tolerance returns the indices of the gates for the current radars

**Parameters**

**radar1, radar2** [radar objects] the two radar objects

**rad1\_ele, rad1\_azimuth, rad1\_rng** [array of floats] the radar coordinates of the radar1 gates

**rad2\_ele, rad2\_azimuth, rad2\_rng** [array of floats] the radar coordinates of the radar2 gates

**ele\_tol, azi\_tol** [floats] azimuth and elevation angle tolerance [deg]

**rng\_tol** [float] range Tolerance [m]

**Returns**

**ind\_ray\_rad1, ind\_rng\_rad1, ind\_ray\_rad2, ind\_rng\_rad2** [array of ints] the ray and range indexes of each radar gate

`pyrad.util.radar_utils.find_contiguous_times(times, step=600)`

Given and array of ordered times, find those contiguous according to a maximum time step

**Parameters**

**times** [array of datetimes] The array of times

**step** [float] The time step [s]

**Returns**

**start\_times, end\_times** [array of date times] The start and end of each consecutive time period

`pyrad.util.radar_utils.find_nearest_gate(radar, lat, lon, latlon_tol=0.0005)`

Find the radar gate closest to a lat,lon point

**Parameters**

**radar** [radar object] the radar object

**lat, lon** [float] The position of the point

**latlon\_tol** [float] The tolerance around this point

**Returns**

**ind\_ray, ind\_rng** [int] The ray and range index

**azi, rng** [float] the range and azimuth position of the gate

`pyrad.util.radar_utils.find_neighbour_gates(radar, azi, rng, delta_azi=None, delta_rng=None)`

Find the neighbouring gates within +-delta\_azi and +-delta\_rng



**Parameters**

**radar** [radar object] the radar object  
**azi, rng** [float] The azimuth [deg] and range [m] of the central gate  
**delta\_azi, delta\_rng** [float] The extend where to look for

**Returns**

**inds\_ray\_aux, ind\_rng\_aux** [int] The indices (ray, rng) of the neighbouring gates

`pyrad.util.radar_utils.find_ray_index(ele_vec, azi_vec, ele, azi, ele_tol=0.0, azi_tol=0.0, nearest='azi')`

Find the ray index corresponding to a particular elevation and azimuth

**Parameters**

**ele\_vec, azi\_vec** [float arrays] The elevation and azimuth data arrays where to look for  
**ele, azi** [floats] The elevation and azimuth to search  
**ele\_tol, azi\_tol** [floats] Tolerances [deg]  
**nearest** [str] criteria to define wich ray to keep if multiple rays are within tolerance. azi: nearest azimuth, ele: nearest elevation

**Returns**

**ind\_ray** [int] The ray index

`pyrad.util.radar_utils.find_rng_index(rng_vec, rng, rng_tol=0.0)`

Find the range index corresponding to a particular range

**Parameters**

**rng\_vec** [float array] The range data array where to look for  
**rng** [float] The range to search  
**rng\_tol** [float] Tolerance [m]

**Returns**

**ind\_rng** [int] The range index

`pyrad.util.radar_utils.get_ROI(radar, fieldname, sector)`

filter out any data outside the region of interest defined by sector

**Parameters**

**radar** [radar object] the radar object where the data is  
**fieldname** [str] name of the field to filter  
**sector** [dict] a dictionary defining the region of interest

**Returns**

**roi\_flag** [ndarray] a field array with ones in gates that are in the Region of Interest

`pyrad.util.radar_utils.get_closest_solar_flux(hit_datetime_list, flux_datetime_list, flux_value_list)`

finds the solar flux measurement closest to the sun hit

**Parameters**

**hit\_datetime\_list** [datetime array] the date and time of the sun hit  
**flux\_datetime\_list** [datetime array] the date and time of the solar flux measurement

**flux\_value\_list:** ndarray 1D the solar flux values

#### Returns

**flux\_datetime\_closest\_list** [datetime array] the date and time of the solar flux measurement closest to sun hit

**flux\_value\_closest\_list** [ndarray 1D] the solar flux values closest to the sun hit time

`pyrad.util.radar_utils.get_data_along_azi` (*radar, field\_name, fix\_ranges, fix\_elevations, rng\_tol=50.0, ang\_tol=1.0, azi\_start=None, azi\_stop=None*)

Get data at particular (ranges, elevations)

#### Parameters

**radar** [radar object] the radar object where the data is

**field\_name** [str] name of the field to filter

**fix\_ranges, fix\_elevations:** list of floats List of ranges [m], elevations [deg] couples

**rng\_tol** [float] Tolerance between the nominal range and the radar range [m]

**ang\_tol** [float] Tolerance between the nominal angle and the radar angle [deg]

**azi\_start, azi\_stop:** float Start and stop azimuth angle of the data [deg]

#### Returns

**xvals** [list of float arrays] The ranges of each rng, ele pair

**yvals** [list of float arrays] The values

**valid\_rng, valid\_ele** [float arrays] The rng, ele pairs

`pyrad.util.radar_utils.get_data_along_ele` (*radar, field\_name, fix\_ranges, fix\_azimuths, rng\_tol=50.0, ang\_tol=1.0, ele\_min=None, ele\_max=None*)

Get data at particular (ranges, azimuths)

#### Parameters

**radar** [radar object] the radar object where the data is

**field\_name** [str] name of the field to filter

**fix\_ranges, fix\_azimuths:** list of floats List of ranges [m], azimuths [deg] couples

**rng\_tol** [float] Tolerance between the nominal range and the radar range [m]

**ang\_tol** [float] Tolerance between the nominal angle and the radar angle [deg]

**ele\_min, ele\_max:** float Min and max elevation angle [deg]

#### Returns

**xvals** [list of float arrays] The ranges of each rng, ele pair

**yvals** [list of float arrays] The values

**valid\_rng, valid\_ele** [float arrays] The rng, ele pairs

`pyrad.util.radar_utils.get_data_along_rng` (*radar, field\_name, fix\_elevations, fix\_azimuths, ang\_tol=1.0, rmin=None, rmax=None*)

Get data at particular (azimuths, elevations)

#### Parameters

**radar** [radar object] the radar object where the data is

**field\_name** [str] name of the field to filter

**fix\_elevations, fix\_azimuths: list of floats** List of elevations, azimuths couples [deg]

**ang\_tol** [float] Tolerance between the nominal angle and the radar angle [deg]

**rmin, rmax: float** Min and Max range of the obtained data [m]

#### Returns

**xvals** [list of float arrays] The ranges of each azi, ele pair

**yvals** [list of float arrays] The values

**valid\_azi, valid\_ele** [float arrays] The azi, ele pairs

`pyrad.util.radar_utils.get_fixed_rng_data` (*radar, field\_names, fixed\_rng, rng\_tol=50.0, ele\_min=None, ele\_max=None, azi\_min=None, azi\_max=None*)

Creates a 2D-grid with (azi, ele) data at a fixed range

#### Parameters

**radar** [radar object] The radar object containing the data

**field\_name** [str] The field name

**fixed\_rng** [float] The fixed range [m]

**rng\_tol** [float] The tolerance between the nominal range and the actual radar range [m]

**ele\_min, ele\_max, azi\_min, azi\_max** [float or None] The limits of the grid [deg]. If None the limits will be the limits of the radar volume

#### Returns

**radar** [radar object] The radar object containing only the desired data

`pyrad.util.radar_utils.get_histogram_bins` (*field\_name, step=None*)  
gets the histogram bins using the range limits of the field as defined in the Py-ART config file.

#### Parameters

**field\_name: str** name of the field

**step** [float] size of bin

#### Returns

**bin\_edges** [float array] The bin edges

`pyrad.util.radar_utils.get_range_bins_to_avg` (*rad1\_rng, rad2\_rng*)

Compares the resolution of two radars and determines if and which radar has to be averaged and the length of the averaging window

#### Parameters

**rad1\_rng** [array] the range of radar 1

**rad2\_rng** [datetime] the range of radar 2

#### Returns

**avg\_rad1, avg\_rad2** [Boolean] Booleans specifying if the radar data has to be average in range

**avg\_rad\_lim** [array with two elements] the limits to the average (centered on each range gate)

`pyrad.util.radar_utils.get_target_elevations(radar_in)`

Gets RHI target elevations

#### Parameters

**radar\_in** [Radar object] current radar object

#### Returns

**target\_elevations** [1D-array] Azimuth angles

**el\_tol** [float] azimuth tolerance

`pyrad.util.radar_utils.join_time_series(t1, val1, t2, val2, dropnan=False)`

joins time\_series. Only of package pandas is available otherwise returns None.

#### Parameters

**t1** [datetime array] time of first series

**val1** [float array] value of first series

**t2** [datetime array] time of second series

**val2** [float array] value of second series

**dropnan** [boolean] if True remove NaN from the time series

#### Returns

**t\_out\_vec** [datetime array] the resultant date time after joining the series

**val1\_out\_vec** [float array] value of first series

**val2\_out\_vec** [float array] value of second series

`pyrad.util.radar_utils.project_to_vertical(data_in, data_height, grid_height, interp_kind='none', fill_value=-9999.0)`

Projects radar data to a regular vertical grid

#### Parameters

**data\_in** [ndarray 1D] the radar data to project

**data\_height** [ndarray 1D] the height of each radar point

**grid\_height** [ndarray 1D] the regular vertical grid to project to

**interp\_kind** [str] The type of interpolation to use: 'none' or 'nearest'

**fill\_value** [float] The fill value used for interpolation

#### Returns

**data\_out** [ndarray 1D] The projected data

`pyrad.util.radar_utils.quantize_field(field, field_name, step)`

quantizes data

#### Parameters

**field** [ndarray 2D] the radar field

**field\_name: str** name of the field

**step** [float] size of bin

**Returns**

**fieldq** [ndarray 2D] The quantized field

**values** [float array] values at each bin

`pyrad.util.radar_utils.rainfall_accumulation(t_in_vec, val_in_vec, cum_time=3600.0, base_time=0.0, dropnan=False)`

Computes the rainfall accumulation of a time series over a given period

**Parameters**

**t\_in\_vec** [datetime array] the input date and time array

**val\_in\_vec** [float array] the input values array [mm/h]

**cum\_time** [int] accumulation time [s]

**base\_time** [int] base time [s]

**dropnan** [boolean] if True remove NaN from the time series

**Returns**

**t\_out\_vec** [datetime array] the output date and time array

**val\_out\_vec** [float array] the output values array

**np\_vec** [int array] the number of samples at each period

`pyrad.util.radar_utils.time_avg_range(timeinfo, avg_starttime, avg_endtime, period)`

finds the new start and end time of an averaging

**Parameters**

**timeinfo** [datetime] the current volume time

**avg\_starttime** [datetime] the current average start time

**avg\_endtime: datetime** the current average end time

**period: float** the averaging period

**Returns**

**new\_starttime** [datetime] the new average start time

**new\_endtime** [datetime] the new average end time

`pyrad.util.radar_utils.time_series_statistics(t_in_vec, val_in_vec, avg_time=3600, base_time=1800, method='mean', dropnan=False)`

Computes statistics over a time-averaged series. Only of package pandas is available otherwise returns None

**Parameters**

**t\_in\_vec** [datetime array] the input date and time array

**val\_in\_vec** [float array] the input values array

**avg\_time** [int] averaging time [s]

**base\_time** [int] base time [s]

**method** [str] statistical method

**dropnan** [boolean] if True remove NaN from the time series

**Returns**

**t\_out\_vec** [datetime array] the output date and time array  
**val\_out\_vec** [float array] the output values array

## PYRAD.UTIL.STAT\_UTILS

Miscellaneous functions dealing with statistics

---

<i>quantiles_weighted</i> (values[, ...])	weight_vector,	Given a set of values and weights, compute the weighted quantile(s) and average.
--	----------------	--

---

```
pyrad.util.stat_utils.quantiles_weighted(values, weight_vector=None, quantiles=array([0.5]), weight_threshold=None, data_is_log=False, nvalid_min=3)
```

Given a set of values and weights, compute the weighted quantile(s) and average.

### Parameters

- values** [array of floats] Array containing the values. Can be 2-dimensional
- weight\_vector** [array of floats or None] array containing the weights to apply. If None it will be an array of ones (uniform weight). If values is a 2D array it will be repeated for the second dimension
- quantiles** [array of floats] The quantiles to be computed
- weight\_threshold** [float or None] If weight\_threshold is set quantiles will be computed only if the total weight (sum of the weights of valid data) exceeds this threshold
- data\_is\_log** [Bool] If true the values will be considered to be in logarithmic scale and transformed into linear scale before computing the quantiles and average
- nvalid\_min** [int] Minimum number of valid points to consider the computation valid

### Returns

- avg** [float] the weighted average
- quants** [array of floats] an array containing the weighted quantiles in the same order as the quantiles vector
- nvalid** [int] Number of valid points in the computation of the statistics





## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### p

pyrad.flow.flow\_aux, ??  
pyrad.flow.flow\_control, ??  
pyrad.graph.plots, ??  
pyrad.graph.plots\_aux, ??  
pyrad.graph.plots\_grid, ??  
pyrad.graph.plots\_spectra, ??  
pyrad.graph.plots\_timeseries, ??  
pyrad.graph.plots\_vol, ??  
pyrad.io.config, ??  
pyrad.io.io\_aux, ??  
pyrad.io.mxpole\_config, ??  
pyrad.io.read\_data\_cosmo, ??  
pyrad.io.read\_data\_dem, ??  
pyrad.io.read\_data\_hzt, ??  
pyrad.io.read\_data\_mxpole, ??  
pyrad.io.read\_data\_other, ??  
pyrad.io.read\_data\_radar, ??  
pyrad.io.read\_data\_sensor, ??  
pyrad.io.read\_data\_sun, ??  
pyrad.io.timeseries, ??  
pyrad.io.trajectory, ??  
pyrad.io.write\_data, ??  
pyrad.proc.process\_aux, ??  
pyrad.proc.process\_calib, ??  
pyrad.proc.process\_cosmo, ??  
pyrad.proc.process\_dem, ??  
pyrad.proc.process\_Doppler, ??  
pyrad.proc.process\_echoclass, ??  
pyrad.proc.process\_grid, ??  
pyrad.proc.process\_intercomp, ??  
pyrad.proc.process\_iq, ??  
pyrad.proc.process\_monitoring, ??  
pyrad.proc.process\_phase, ??  
pyrad.proc.process\_retrieve, ??  
pyrad.proc.process\_spectra, ??  
pyrad.proc.process\_timeseries, ??  
pyrad.proc.process\_traj, ??  
pyrad.prod.process\_grid\_products, ??  
pyrad.prod.process\_intercomp\_products, ??  
pyrad.prod.process\_monitoring\_products, ??  
pyrad.prod.process\_product, ??  
pyrad.prod.process\_spectra\_products, ??  
pyrad.prod.process\_timeseries\_products, ??  
pyrad.prod.process\_traj\_products, ??  
pyrad.prod.process\_vol\_products, ??  
pyrad.prod.product\_aux, ??  
pyrad.util.radar\_utils, ??  
pyrad.util.stat\_utils, ??