HSR1 Python library

## Introduction

This manual describes a Python library for analysing datasets from the HSR1 radiometer and displaying a summary of the data collected. This comes in two parts:

1. For early instruments, the text-based data files are loaded and stored into a database in the same format used by later instruments.
2. For datasets in the database format, the library can then quickly plot summary charts showing the complete dataset, or show data in more detail in daily plots. These give an overview of the health and accuracy of the readings, including the GPS and other internal sensors.

Version

This manual describes version 1.1.0 of the python library

## Authors

Albie Gray, John Wood

Copyright 1 July 2024



Contents

[Introduction 1](#_Toc173925729)

[Version 1](#_Toc173925730)

[Authors 1](#_Toc173925731)

[Installation 3](#_Toc173925732)

[pip 3](#_Toc173925733)

[Timezones 4](#_Toc173925734)

[HSR1 Timezone settings 4](#_Toc173925735)

[Database timezone settings 4](#_Toc173925736)

[Angles 4](#_Toc173925737)

[Flags 4](#_Toc173925738)

[Wavelength calibration 5](#_Toc173925739)

[Examples 5](#_Toc173925740)

[Storing .txt files to the database 5](#_Toc173925741)

[Loading data from the database 5](#_Toc173925742)

[Making graphs from a database 6](#_Toc173925743)

[Calculating and graphing aod 7](#_Toc173925744)

[Reference 8](#_Toc173925745)

[DBDriver 8](#_Toc173925746)

[add\_precalculated\_values 9](#_Toc173925747)

[make\_gpx 9](#_Toc173925748)

[store 9](#_Toc173925749)

[store\_raw 10](#_Toc173925750)

[combine\_database 10](#_Toc173925751)

[combine\_database\_folder 11](#_Toc173925752)

[load 11](#_Toc173925753)

[load\_metadata 13](#_Toc173925754)

[load\_accessory 13](#_Toc173925755)

[load\_table\_names 14](#_Toc173925756)

[load\_spectrum 14](#_Toc173925757)

[load\_raw 15](#_Toc173925758)

[load\_sql 16](#_Toc173925759)

[Graph 16](#_Toc173925760)

[plot\_integral 18](#_Toc173925761)

[plot\_gps 19](#_Toc173925762)

[plot\_accessory 21](#_Toc173925763)

[plot\_dips\_summary 22](#_Toc173925764)

[plot\_daily\_line 23](#_Toc173925765)

[plot\_daily\_hists 24](#_Toc173925766)

[biggest\_dips 25](#_Toc173925767)

[daily\_biggest\_dips 26](#_Toc173925768)

[plot\_time\_day 27](#_Toc173925769)

[Elevation/azimuth plot 28](#_Toc173925770)

[plot\_spectrum\_day 29](#_Toc173925771)

[plot\_aod\_day 30](#_Toc173925772)

[wait\_until\_closed 33](#_Toc173925773)

[read\_txt 33](#_Toc173925774)

[read 33](#_Toc173925775)

[read\_raw\_txt 34](#_Toc173925776)

[read\_raw\_pixels 34](#_Toc173925777)

[utils.HsrFunc 34](#_Toc173925778)

[calc\_direct\_normal\_spectrum 35](#_Toc173925779)

[calc\_sun\_zenith 35](#_Toc173925780)

[calc\_rayleigh 35](#_Toc173925781)

[calc\_air\_mass 35](#_Toc173925782)

[calc\_aot\_direct 36](#_Toc173925783)

[calc\_cimel\_band\_aot\_direct 36](#_Toc173925784)

[calc\_aod\_from\_df 37](#_Toc173925785)

[load\_et\_spectrum 38](#_Toc173925786)

[calculate\_clearsky\_filter 38](#_Toc173925787)

[Database reference 38](#_Toc173925788)

[spectral\_data 38](#_Toc173925789)

[system\_data 39](#_Toc173925790)

[precalculated\_values 40](#_Toc173925791)

[deployment\_metadata 40](#_Toc173925792)

[accessory\_data 42](#_Toc173925793)

[raw\_data 43](#_Toc173925794)

[Deployment.ini file 43](#_Toc173925795)

[License 44](#_Toc173925796)

# Installation

## pip

This library can be installed using pip with the command:

pip install hsr1

And updated to the latest version using this command:

pip install --upgrade hsr1

# Timezones

## HSR1 Timezone settings

Reading times are recorded by the instrument using the Windows time setting. It is recommended to set this to UTC, though other timezones are acceptable. It is best to avoid any daylight saving, which can change the timezone part way through a dataset.

The HSR1 timezone is read from the deployment.ini file, which must be created by the user on early HSR1s. This also contains other metadata about the data series and datalogging settings. There is an example of this file in the deployment metadata section at the end of this manual.

## Database timezone settings

Reading times are stored in the database in local time as a timezone aware timestamp. If there are multiple dataseries in a database with different timezones (for example during daylight saving time) it will work the same way, with each measurement being in the local time when it was measured, according to the user supplied timezone.

When loading from the database, all timezones will be loaded into Python as a timezone aware pandas timestamp, which is mostly interchangeable with a python datetime.datetime. The timezone used on loading is passed in the db\_driver.load() command, and may be different from the timezone used in recording the dataset. In general, UTC is used for all astronomical calculations, and the local timezone is used in presenting the data where a local time is appropriate, e.g. in separating data into daily chunks.

# Angles

Internally, all angles are stored as radians and all functions that take an angle take it in radians. When loading data from the database, angle columns will be in radians. All graphs are in degrees

# Flags

Included in the graphing section of the library is the capability to detect implausible data points and overlay them in a different colour. This means that potentially misleading datapoints are easy to spot and can’t be confused with real data. Currently the only graphs that display flagged data are the graphs in the [integral summary](#_plot_integral), with the parameter flag=True, and daily line plots when plotting integral data, again with the parameter flag=True. In future versions of this library, it will be possible to automatically filter out the flagged readings. You can, however, calculate the flags manually using this code:

from hsr1.plots import flagData

flags = flagData.flag(data, ignore\_nights=True)

Where data is a dataframe with columns: ["global\_integral", "diffuse\_integral", "direct\_normal\_integral", "sza", "toa\_hi"]

The flags dataframe can then be used to filter the data returned from the database

# Wavelength calibration

This library can also calibrate the wavelengths of a hsr1 instrument from solar measurements. Instructions on how to do this are in the Wavelength Calibration user manual.

# Examples

## Storing .txt files to the database

1. Set the locations of the data filepath, the Deployment.ini file, and the database to create or update.
2. Create a DBDriver object with the location of the database (if it doesn’t exist, it will create a new one).
3. Read the data from the .txt files, and the Deployment.ini file.
4. Store the data to the database. During this process, the solar position information (solar azimuth & zenith, atmospheric path length) will be calculated and stored to the database.

import hsr1

data\_filepath = "../Datasets/Small Tara 2023"

deployment\_metadata\_filepath = "../Datasets/Tara 2023/Tara 2023 Deployment.ini"

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

data = hsr1.read\_txt.read(data\_filepath, deployment\_metadata\_filepath=

deployment\_metadata\_filepath)

db\_driver.store(data)

## Loading data from the database

1. Make a DBDriver object.
2. Use db\_driver.load(), passing in a list of all the columns you want, start time, end time and any SQL conditions.

import hsr1

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

data = db\_driver.load(["pc\_time\_end\_measurement", "global\_integral"],

condition="sza < 1.57")

This example will load the time and the global irradiance for all records where the sun is above the horizon.

This example demonstrates how data from some tables (like spectral\_data and precalculated\_values) can be loaded together, but other tables, like accessory\_data, have to be loaded separately

import hsr1

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

global\_spectrum = db\_driver.load\_spectrum("global\_spectrum")

import hsr1

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

# data from accessory\_data

accessory\_data = db\_driver.load\_accessory(["pc\_time\_end\_measurement", "Yaw"])

# data from spectral\_data and precalculated\_values

data = db\_driver.load(["global\_integral", "azimuth"])

This example demonstrates loading spectral data from the database

## Making graphs from a database

1. Make a graph object, passing in a DBDriver or the name of a database.
2. Use the methods of the graph object to plot the graphs you want.
3. Some graphs are more generic, and you can pass in the parameters you want to plot, these are detailed in the [reference](#_vmgxekh30cy4) section.

import hsr1

database\_location = "databases/my\_database.db"

graph = hsr1.Graph(database\_location)

graph.plot\_integral()

graph.plot\_gps()

graph.plot\_dips\_summary()

graph.plot\_accessory()

graph.plot\_daily\_line(["global\_integral", "diffuse\_integral"])

graph.plot\_daily\_hist(["rh", "pressure"])

## Calculating and graphing aod

This is an example of how to make a line graph of the atmospheric optical depth at 5 different wavelengths. This covers loading from the database, unpacking spectral data, and using the existing graphs.

# import hsr1, and also numpy and pandas which are used for data processing

import numpy as np

import pandas as pd

import hsr1

# creates DBDriver object for the database where the data to be plotted is stored

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

# loads the data that will be used to calculate the aod

data = db\_driver.db\_load.load(["pc\_time\_end\_measurement", "sza", "sed"])

global\_spectrum = db\_driver.load\_spectrum("global\_spectrum")

diffuse\_spectrum = db\_driver.load\_spectrum("diffuse\_spectrum")

# only the aod\_microtops will be plotted

aod\_type = "aod\_microtops"

# calculates the aod values that will be plotted.

aod\_data = hsr1.utils.HsrFunc.calc\_aot\_direct(global\_spectrum, diffuse\_spectrum,

pd.DataFrame(data["sza"]),

sed=data["sed"], aod\_type=aod\_type)

# these are the wavelengths that will be plotted

wavelengths = np.array([380, 440, 500, 675, 870, 1020])

# filters for just the aod type that is being used, and converts the series of arrays into a 2d array

aod = np.stack(aod\_data[aod\_type].values)

# filters the wavelengths to only include the selected wavelengths.

# -300 because the wavelengths are from 300-1100, so the first in the array is 300nm

aod = aod[:, wavelengths-300]

# convert the array back into a dataframe, and add the time column

limited\_df = pd.DataFrame(aod, columns=wavelengths.astype(str))

limited\_df["pc\_time\_end\_measurement"] = aod\_data["pc\_time\_end\_measurement"]

# calculates which readings have a clear sky and which are cloudy. uses the "wood" method by default

# calculate\_clearsky\_filter returns an array of 1s and 0s which can be used to filter an array or dataframe

clearsky\_filter = hsr1.utils.HsrFunc.calculate\_clearsky\_filter(data, global\_spectrum, diffuse\_spectrum)

# filter the aod and time values using the clearsky filter

# this will cause gaps in the index where values were skipped,

# reset\_index returns it to a normal increasing index

limited\_df = limited\_df.loc[clearsky\_filter, :].reset\_index(drop=True)

# creates a graph object

graph = hsr1.Graph(db)

# plots the graph

# sets rows=3, days\_in\_row=7 to put fewer days on each page, which shows each day

# in more detail than putting a whole month on a page

graph.plot\_daily\_line(dataframe=limited\_df, columns=wavelengths.astype(str),

rows=3, days\_in\_row=7, title\_prefix=aod\_type+"\n")

# Reference

## DBDriver

hsr1.DBDriver(db\_name, db\_type="sqlite")

Generic class for database operations.

DBDriver contains a [Load](#_Load) and a [Store](#_Store) object, and exposes common methods in those objects to the user for easy access.

Example:

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

Parameters:

* db\_name(str): name of the db. if the db doesn't exist, it will be created when data is stored to it.
* db\_type(str): type of database used. Currently only sqlite is supported.

Attributes:

* db\_name(str) the name of the database
* db\_store(hsr1.db.Store): a [Store](#_Store) object connected to this database
* db\_load(hsr1.db.Load): a [Load](#_Load) object connected to this database
* last\_timestamp(str): the last timestamp in the database
* first\_timestamp(str): the first timestamp in the database

Methods:

* [add\_precalculated\_values](#_add_precalculated_values)
* [make\_gpx](#_make_gpx)
* [store](#_store_1)
* [store\_raw](#_store_raw)
* [combine\_database](#_combine_database_1)
* [combine\_database\_folder](#_combine_database_folder_1)
* [load](#_load_1)
* [load\_metadata](#_load_metadata)
* [load\_accessory](#_load_accessory)
* [load\_table\_names](#_load_table_names)
* [load\_spectrum](#_load_spectrum)
* [load\_raw](#_load_raw)
* [load\_sql](#_load_sql_1)

### add\_precalculated\_values

db\_driver.add\_precalculated\_values()

calculates some commonly used values and stores them to the database. These include solar azimuth & zenith angles, solar irradiance at top of atmosphere, and airmass.

Running this with no parameters passed will just recalculate the precalculated\_values for the whole dataset.

Parameters:

* method(str): which library to use.

"ephem": basic calculation

"sg2\_static": really fast for a static instrument

"sg2\_mobile": slow, not recommended, but works for a mobile dataset.

"sg2": detects whether the dataset is mobile and uses the appropriate method.

None(default): uses "sg2\_static" if static, and "ephem" if the dataset is mobile.

* mobile: wether or not the dataset's location will change. if you know it won’t, you can pass mobile=False and calculations will be sped up
* sample\_ids\_to\_add(np.array): array of sample ids that correspond to the readings that precaclulated\_values will be calculated for
* drop\_existing: deletes the existing data in the database. default true as this function is usually used to recalculate a whole dataset. To fill in a gap or recalculate a small section and keep the rest, set this to False and pass in the sample ids you want to add

Returns:

* None, just updates the database with a new table.

### make\_gpx

db\_driver.make\_gpx(filename)

make and save an .xml file, containing the latitude & longitude values and timestamp.

The file can be used with google my maps or another gpx viewer.

Parameters:

* filename(str): the file location that the .gpx file will be saved to

Returns:

* None

### store

db\_driver.store(dfs, precalculate=True)

stores data to the database

This function should usually only be used straight after reading data from a dataset of .zip files, and you can pass the output of that straight into the dfs parameter of this function.

Parameters:

* dfs(tuple(pandas.dataframe)): tuple of 3 or 4 dataframes each containing data for a different table of the database.
  + The order is: spectral\_data, system\_data, deployment\_metadata, accessory\_data (if included)
* precalculate(bool): if True, calculates some commonly used values and stores them to a separate table in the database. These are required for some graphs, so it is recommended to keep this as True.

### store\_raw

db\_driver.store\_raw(dfs, deployment\_metadata)

stores raw data to the database

parameters:

dfs: tuple of dataframes, one dataframe per channel

deployment\_metadata: dataframe with one row, containing the deployment metadata for the dataseries being stored

If no raw data already exists, creates a new table, "raw\_data" which stores spectral data like in the spectral\_data table, with each reading containing a blob that can be decoded to a spectral array.

### combine\_database

db\_driver.combine\_database(new\_db\_name)

combines another database into the current database.

Parameters:

* new\_db\_name(str): file path to the database that will be merged into the current one.

The current database is the one connected to the [DBDriver](#_DBDriver) that you are accessing store from. This function will append all the data in the new database onto the current database.

Example:

import hsr1

existing\_database = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(existing\_database)

db\_driver.combine\_database("databases/my\_other\_database.db")

### combine\_database\_folder

db\_driver.combine\_database\_folder(folder, delete=False)

Combines a folder of databases into the current database.

Parameters:

* folder(str): the folder containing several databases to merge into the current database.
* delete(bool): if True, deletes the contents of the the folder after they are combined

Calls combine\_database on each database in the folder.

### load

db\_driver.load(columns=[], table=None, start\_time=None, end\_time=None,

condition="", raise\_on\_missing=True, sort=True,

deserialise=True, timezone="+00:00")

Loads requested columns from the database.

Parameters:

* columns(list(str)): default []. list of database columns that you want data from. If [] then all columns in spectral\_data, system\_data, and precalculated\_values will be returned. See the note about loading accessory\_data and data from other tables. List of all the database tables and columns can be found [here](#DatabaseReferenceTitle).
* table(str): the table you are loading data from. "spectral\_data", "system\_data", "precalculated\_values", "deployment\_metadata" and if your database has it, "accessory\_data". Use this if the columns you want are in multiple tables and you want to specify which one you want.
* start\_time(str): string representing a timestamp. Format is YYYY-MM-DD HH:MM:SS in the device’s local time that is stored in the database, not in the timezone that is passed in the timezone parameter. Timestamps are stored as strings in the database, so there is a simple string comparison. You don't need to provide a full timestamp, it will return all timestamps after your query and is inclusive.
* end\_time(str): see above.
* condition(str): for more complex queries. A string that will be appended to the SQL query. " WHERE " is automatically appended so no need to include that. For even more control, use [load\_sql](#_wwbhnu5cgpk4)
  + E.g.: "sza < 1.57" will select all values where sza is less than 1.57.
* raise\_on\_missing(bool): default True. If one or more columns couldn’t be found in the database, the program will raise an error. Set this to False to skip missing columns
* sort(bool): default True. Sorts the result by the database’s local reading time. This doesn't take long on most databases, but if you have a large database that you know is sorted, or doesn't need to be sorted, setting this to false can speed up query times.
* deserialise(bool): default True. Automatically deserialise the spectral data back into numpy arrays.
* timezone(str): default "+00:00". The time zone that all the data will be converted into when loaded. This ensures all data is in the same time zone when it is loaded from the database

Returns:

* result(pandas.dataframe): one dataframe with all requested database columns as columns.

Note: because some tables have different reading frequencies (deployment\_metadata will only have a handful of readings, one for each data series, accessory\_data has readings at the sampling frequency, and all the other tables have readings at the storage (averaging) frequency) you can't load columns from multiple tables in the same request, if they have different measurement periods. To get around this, do separate calls to load for each table type.

Note: spectral data columns, e.g. global\_spectrum is also returned as a single column in the output dataframe, each reading is a numpy array of 801 floats representing the values at wavelengths 300-1100. To convert from the column of arrays in to one dataframe with 801 columns, run this code snippet:

new\_df = pd.DataFrame(np.stack(total\_dataframe[spectral\_column].values),

columns=np.arange(300, 1101, 1))

Note: if table is not specified and a requested column is in multiple tables, this method will try and guess which one you want. The priorities are below:

1. All columns are in the same table type (if you request pc\_time\_end\_measurement and some accessory\_data columns, it will return the time in the accessory\_data table), to avoid a crash.
2. The table priorities when there is no other way to distinguish between tables:
   1. spectral\_data
   2. system\_data and precalculated\_values (same priority as there are no shared columns that aren't in spectral\_data)
   3. accessory\_data and deployment\_metadata (same priority as there are no shared columns that aren't in spectral\_data)

Example:

import hsr1

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

data = db\_driver.load(["pc\_time\_end\_measurement", "global\_integral"],

condition="sza < 1.57")

### load\_metadata

db\_driver.load\_metadata(columns=[], condition="", raise\_on\_missing=True)

Loads the requested data from the deployment\_metadata table.

Parameters:

* columns(list(str)): default []. List of database columns that you want data from. If [] then all columns will be returned.
* condition(str): for more complex queries. A string that will be appended to the SQL query. “ WHERE “ is automatically appended so no need to include that.
  + E.g.: "sza < 1.57" will select all values where sza is less than 1.57.
* raise\_on\_missing(bool): default True. If one or more columns couldn’t be found in the database, the program will raise an error. Set this to False to skip missing columns.

The same as calling [load](#_load_2)() with table="deployment\_metadata", but this method only has the parameters that have an effect on the deployment\_metadata table

example:

import hsr1

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

data = db\_driver.load\_metadata(["owner\_contact", "operator\_contact"])

### load\_accessory

db\_driver.load\_accessory(columns=[], start\_time=None, end\_time=None,

raise\_on\_missing=True, sort=True,

timezone="+00:00")

Loads the requested data from the accessory\_data table.

The same as passing [load](#_load_2)() with table="accessory\_data". Same parameters as load() but without table, as that is always "accessory\_data", and deserialise, as there is no serialised data in this table.

Parameters:

* columns(list(str)): default []. list of database columns that you want data from. If [] then all columns will be returned
* start\_time(str): string representing a timestamp. Format is YYYY-MM-DD HH:MM:SS in local time. Timestamps are stored as strings in the database, so there is a simple string comparison. You don't need to provide a full timestamp, it will return all timestamps after your query and is inclusive.
* end\_time(str): see above.
* condition(str): for more complex queries. A string that will be appended to the SQL query. “ WHERE “ is automatically appended so no need to include that.
  + E.g.: "sza < 1.57" will select all values where sza is less than 1.57.
* raise\_on\_missing(bool): default True. If one or more columns couldn’t be found in the database, the program will raise an error. Set this to False to skip missing columns.
* sort(bool): default True. Sorts the result by the database’s local reading time. This doesn't take long on most databases, but if you have a large database that you know is sorted, or doesn't need to be sorted, setting this to false can speed up query times.
* timezone(str): default "+00:00". The time zone that all the data will be converted into when loaded. This ensures all data is in the same time zone when it is loaded from the database.

example:

import hsr1

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

# data from accessory\_data

accessory\_data = db\_driver.load\_accessory(["pc\_time\_end\_measurement", "Yaw"])

### load\_table\_names

db\_driver.load\_table\_names()

return a dictionary of all the table names and their corresponding column names.

Format: {"table\_1": ["column\_1", "column\_2", "column\_3"], "table\_2":[ "column\_1", …], ...}

Parameters:

* None

Returns:

* Dictionary containing table and column names.

### load\_spectrum

db\_driver.load\_spectrum(column, table=None, start\_time=None,

end\_time=None, condition="", raise\_on\_missing=True, sort=True,

deserialise=True, timezone="+00:00")

loads one spectral column and converts it into a single dataframe with one column per nm and one reading per measurement

* columns(list(str)): default []. list of database columns that you want data from. If [] then all columns in spectral\_data, system\_data, and precalculated\_values will be returned. See the note about loading accessory\_data and data from other tables. List of all the database tables and columns can be found [here](#DatabaseReferenceTitle).
* table(str): the table you are loading data from. "spectral\_data", "system\_data", "precalculated\_values", "deployment\_metadata" and if your database has it, "accessory\_data". Use this if the columns you want are in multiple tables and you want to specify which one you want.
* start\_time(str): string representing a timestamp. Format is YYYY-MM-DD HH:MM:SS in the device’s local time that is stored in the database, not in the timezone that is passed in the timezone parameter. Timestamps are stored as strings in the database, so there is a simple string comparison. You don't need to provide a full timestamp, it will return all timestamps after your query and is inclusive.
* end\_time(str): see above.
* condition(str): for more complex queries. A string that will be appended to the SQL query. " WHERE " is automatically appended so no need to include that. For even more control, use [load\_sql](#_wwbhnu5cgpk4)
  + E.g.: "sza < 1.57" will select all values where sza is less than 1.57.
* raise\_on\_missing(bool): default True. If one or more columns couldn’t be found in the database, the program will raise an error. Set this to False to skip missing columns
* sort(bool): default True. Sorts the result by the database’s local reading time. This doesn't take long on most databases, but if you have a large database that you know is sorted, or doesn't need to be sorted, setting this to false can speed up query times.
* deserialise(bool): default True. Automatically deserialise the spectral data back into numpy arrays.
* timezone(str): default "+00:00". The time zone that all the data will be converted into when loaded. This ensures all data is in the same time zone when it is loaded from the database

example:

import hsr1

database\_location = "databases/my\_database.db"

db\_driver = hsr1.DBDriver(database\_location)

global\_spectrum = db\_driver.load\_spectrum("global\_spectrum")

### load\_raw

db\_driver.load\_raw(columns=[], start\_time=None, end\_time=None)

loads the raw data from the raw\_data table

params:

* columns: list of columns that you want to load, they are in the format: "channel\_0", "channel\_1" etc. If [], all columns are loaded
  + start\_time, end\_time: for only selecting data within a time period. format: ISO-8601("2023-05-23 00:00:00").
  + if you don’t put in a full datetime, it will assume first timestamp, e.g: 2023-05-23 will be read as 2023-05-23 00:00:00. Therefore, start\_time is inclusive, end time is not

returns:

* a dataframe with the requested data. pc\_time\_end\_measurement is automatically added as a column columns = pc\_time\_end\_measurement, requested\_channel\_0, requested\_channel\_1. If only one column is requested, it is returned as one unpacked dataframe, index=time, columns=wavelength

### load\_sql

db\_driver.load\_sql(sql)

Runs a custom SQL query on the database, and returns the result as a dataframe

Parameters:

* sql(str): the query to run.

Returns:

* A dataframe containing the requested data. If the query doesn't return any data, returns None.

## Graph

hsr1.Graph(data=None, driver=None, dataframe=None,

deployment\_metadata=None, accessory\_data=None,

output\_location=None, diffuse\_name="DIF", timezone="+00:00",

block=True, dpi=100,

\*\*kwargs)

A class for graphing HSR1 data.

this contains many examples of how the underlying graph classes in hsr1.plots can be used. there are combined plots with several similar graphs, some functions that provide a simple wrapper around a class, and some that load data, do some data processing, and then use one of the plots from hsr1.plots. if you want to implement your own graph, this is a good place to.

Example:

import hsr1

database\_location = "databases/my\_database.db"

graph = hsr1.Graph(database\_location, start\_time="2024-06-17",

end\_time="2024-06-18", dpi=300, timezone="+06:00",

output\_location="../hsr1\_debug\_images/")

Parameters:

* data(hsr1.DBDriver, pandas.dataframe, str): to be used as a positional argument, as either a DBDriver or a dataframe containing all the data(these can also be passed as keyword arguments), or a string of the database name. if this is a DataFrame, it should contain all the data you want to plot in the spectral\_data, system\_data, and precalculated\_values table of the database
* driver(hsr1.DBDriver): to pass a DBDriver object as a keyword argument.
* dataframe(pandas.dataframe: to pass a dataframe of data as a keyword argument. The data in this dataframe should be from the spectral\_data, system\_data, and precalculated\_values tables in the database, data from the other tables can be passed below.
* deployment\_metadata(pandas.dataframe): to pass a dataframe of deployment metadata as a keyword argument.
* accessory\_data(pandas.dataframe): to pass a dataframe of accessory\_data as a keyword argument.
* output\_location(str): String filepath to a folder where all the graphs generated will be stored.
* diffuse\_name(str): what to call the diffuse on graph labels. Default="DIF"
* timezone(str): the timezone to add to the time in the dataset.
* dpi(int): the dots per inch of the figure, measure of resolution, increase this if the graphs are blurry, decrease this if it is taking too long to generate.
* block(bool): whether matplotlib waits for the user to close a graph before generating the next one.
  + If true, you will have to manually close each graph to make the program run.
  + If false, all the graphs will be plotted as fast as they are generated.
  + note: as python doesn’t wait for the user to close plots before continuing, when it reaches the end of the program, it will terminate, closing all the plots with it. if you want the plots to persist once they're all generated, call graph.wait\_until\_closed()
* Also supports passing keyword arguments into [load](#_7mwxritdxu2m) when loading data from a database.

Attributes

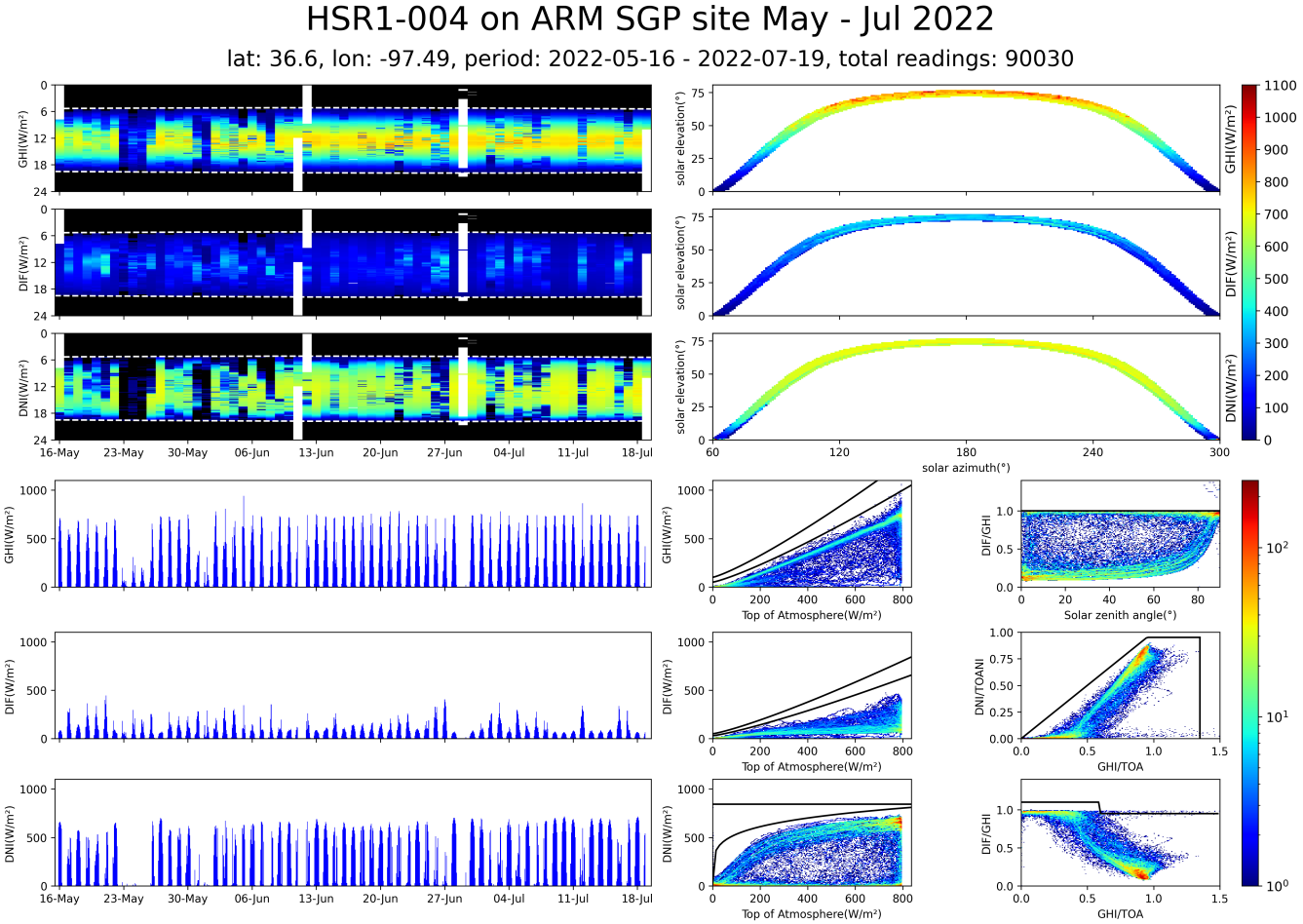
* All the above parameters
* timedelta(pandas.timedelta): the time difference between local time and UTC.

Methods:

* [plot\_integral](#_plot_integral)
* [plot\_gps](#_plot_gps)
* [plot\_dips\_summary](#_plot_dips_summary)
* [plot\_accessory](#_plot_accessory)
* [plot\_daily\_line](#_plot_daily_line)
* [plot\_daily\_hists](#_plot_daily_hists)
* [biggest\_dips](#_biggest_peaks)
* [daily\_biggest\_dips](#_daily_biggest_dips)
* [plot\_time\_day](#_plot_time_day)
* [plot\_spectrum\_day](#_Spectrum_day_plot)
* [plot\_aod\_day](#_AOD_day_plot)
* [daily\_aod\_cimel](#_Daily_AOD_line)
* [wait\_until\_closed](#_wait_until_closed)

### plot\_integral

graph.plot\_integral(flag=False, title="", max\_integral=None)



time/day plots(top left):

Each column represents one day of data, red=brightest, blue=darkest, white=missing data. Dotted white lines represent sunrise and sunset.

elevation/azimuth plots(top right):

Plots the solar elevation against the azimuth, with the colour representing the intensity, same scale as time/day plots. Shows the path of the sun throughout the dataset. Can be used to get a sense of where in the day the sun is brightest, and can also be used to find obstructions to the instrument

Integral plots(bottom left):

Simple plots of the global, diffuse and direct normal intensity over time. On a smaller dataset, you can get a good overview of each day, whereas on a full dataset you can see how the intensity varies over time. These show the broadband values integrated over the range 400nm – 1000nm.

accuracy/plausibility checks(bottom right):

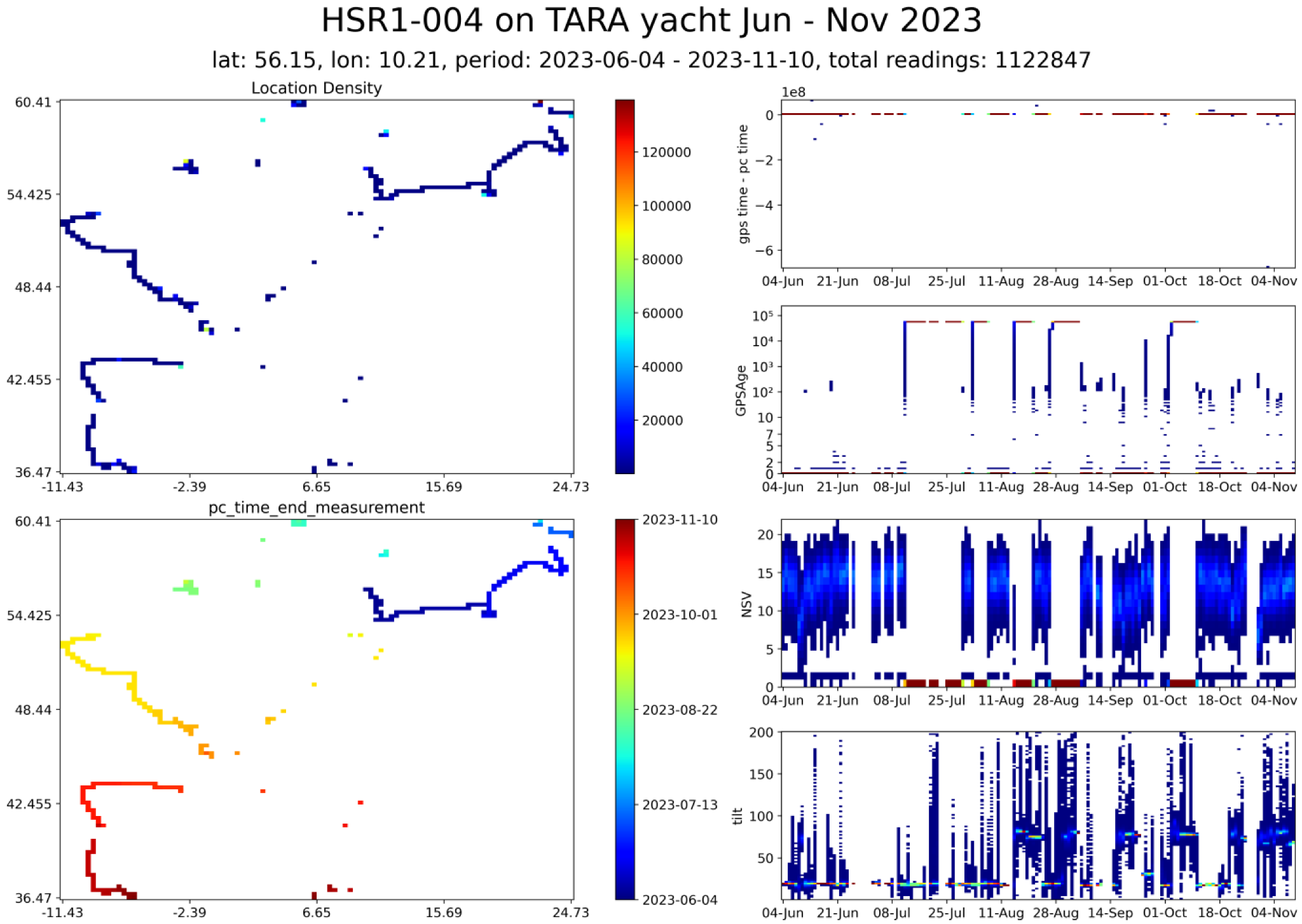
A series of plots that help check the accuracy of the data. The black lines are the BSRN’s Physically possible limits, and the extremely rare limits. When the flag parameter is True (False by default), all data over the Physically possible line will be plotted as pink on all the other graphs in the summary. When flag is False, only the values over the limit for each individual graph will be plotted in pink. Details of what each graph is plotting is on the axes. Colour represents density. NB – the BSRN limit values are reduced to the extraterrestrial spectrum integrated over 400nm – 1000nm to match the HSR1 outputs.

Parameters:

* flag(bool) default False: overlays the implausible data (from the accuracy/plausibility checks) on top of the regular data in pink.
* title(str) default "": the title to display at the top of the plot. It left blank, it will be filled with the title field of the deployment\_metadata table.
* max\_integral(float) default None: the maximum intensity reading. Sets the colour scale and axis ylimits. Automatically set to the largest value in ghi, dhi or dni, but if you want to compare multiple datasets with the same axes and colour scales, pass the same max\_integral here.

### plot\_gps

graph.plot\_gps()



Plots a summary of the gps and related data. On a mobile dataset, shows where the instrument has been over the dataset. On a static dataset, shows the variation of the gps

Gps and frequency plot (top left)

Plots the latitude and longitude, with the colour representing the number of readings at that location

Gps and datetime plot (bottom left):

Plots the latitude and longitude, with the colour representing the time of the most recent reading at that location

Gps time error (top right):

Plots gps time-pc time on a modified logarithmic scale. It is logarithmic above and below 0, and -10 to +10 are linear.

GPSAge (middle top right):

Plots the time (s) since the last valid gps reading. This shows when the gps is not reading correctly.

NSV (middle bottom right):

Plots the number of satellites in view. Low values mean inaccurate gps.

Note: it is common to see NSV values up to 20, which may seem incorrect, as there are only 24 GPS satellites (which means you can only ever see a maximum of 12 at once), but these higher values are because satellites from other constellations are visible, e.g: GLONASS

Tilt (bottom right):

The output of the integrated tilt sensor, calculated as the square root of roll^2 + pitch^2

When the database has a simple GPS file rather than accessory\_data, there are different plots, rather than just empty space:

baro\_temp (middle top right):

The temperature reading inside the instrument from the barometer, replaces GPSAge graph

Pressure (middle bottom right):

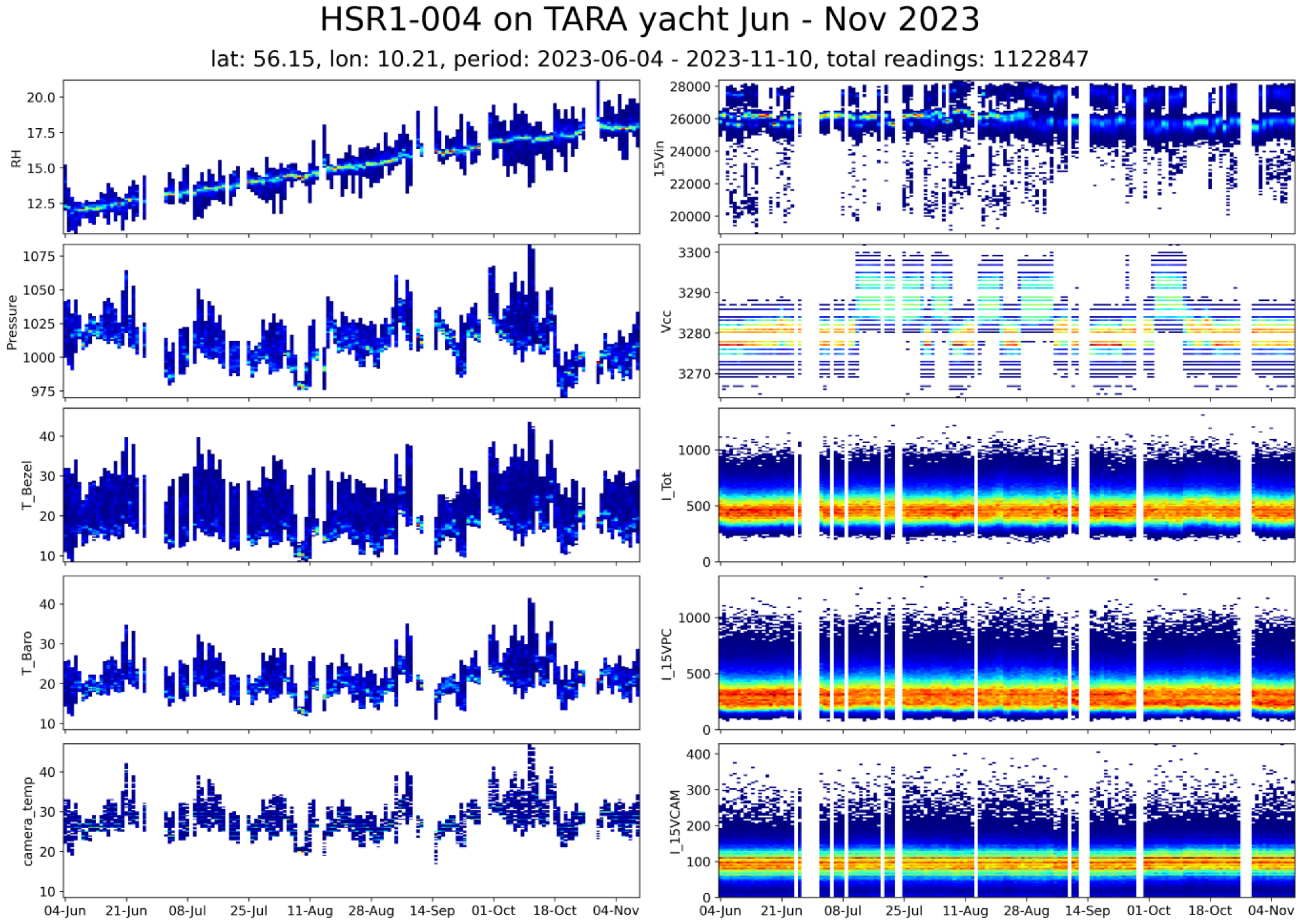
The pressure reading inside the instrument, replaces NSV graph

Relative humidity (bottom right):

The relative humidity inside the instrument, replaces tilt graph

### plot\_accessory

graph.plot\_accessory()



Plots a summary of the data in the accessory\_data table. Will raise an exception if the database does not have an accessory\_data table.

The plots:

Relative humidity inside the enclosure

Pressure inside the enclosure

Temperature at the bezel

Temperature as measured by the barometer

Camera temperature

15Vin – the power supply to the instrument

VCC – the internal CPU supply rail

Total current into the instrument

Current used by the pc

Current used by the camera

### plot\_dips\_summary

graph.plot\_dips\_summary()

A screen shot of a computer

Description automatically generated

Plots the wavelength of the 15 most prominent absorption dips at each reading as a frequency histogram. This can give a picture of how accurate the wavelength calibration is over the whole dataset. The correct locations of some prominent dips are plotted on the y-axis and with a grey line on the graph as a reference

Params:

* n(int) default 15: number of dips to choose from each reading.
  + Dips are chosen by most prominent first.
  + A small n value may miss out some dips, and a large one will be noisy.
* cutoff\_angle(float) default 80: Degrees. Only include solar zenith angles less than this value. Used for only including daylight data.
* cutoff\_wavelength(float) default 1000: only include wavelengths less than this value. Used to avoid noisy readings at the top of the spectrum.
* reference\_lines(list) default shown above: list of wavelengths where dips should appear, will be superimposed onto the graph as a reference.
* reference\_labels(list) default shown above: labels for the reference lines. Must be the same length as reference\_lines, pass "" if you don't want to label a value.

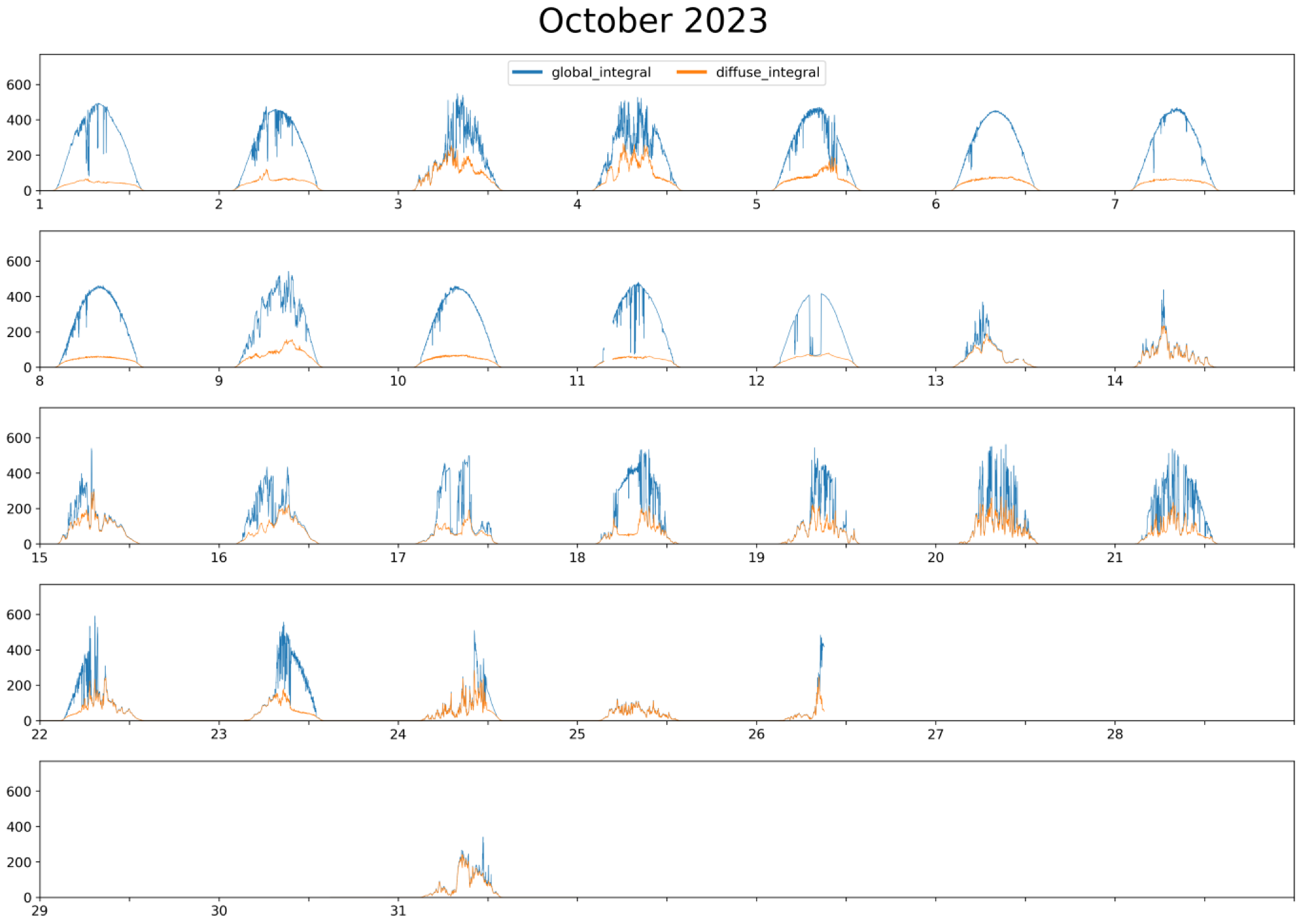
### plot\_daily\_line

graph.plot\_daily\_line(columns, period="monthly", rows=None,

days\_in\_row=None, flag=False, ignore\_zero=False,

title\_prefix="", \*\*kwargs)

plots a line graph of each requested column



Plots a line graph of some data columns, spread over multiple pages if necessary. You can control how the days are spread over the pages using the period, rows, and days\_in\_row parameters.

Note: even if you pass a period value, if you also pass a rows and a days\_in\_row values, they will take precedence and may not show the whole period.

There are some presets with titles and columns already selected:

* daily\_integrals() The global, diffuse, and direct normal
* daily\_temps() All the internal temperatures measured

Parameters:

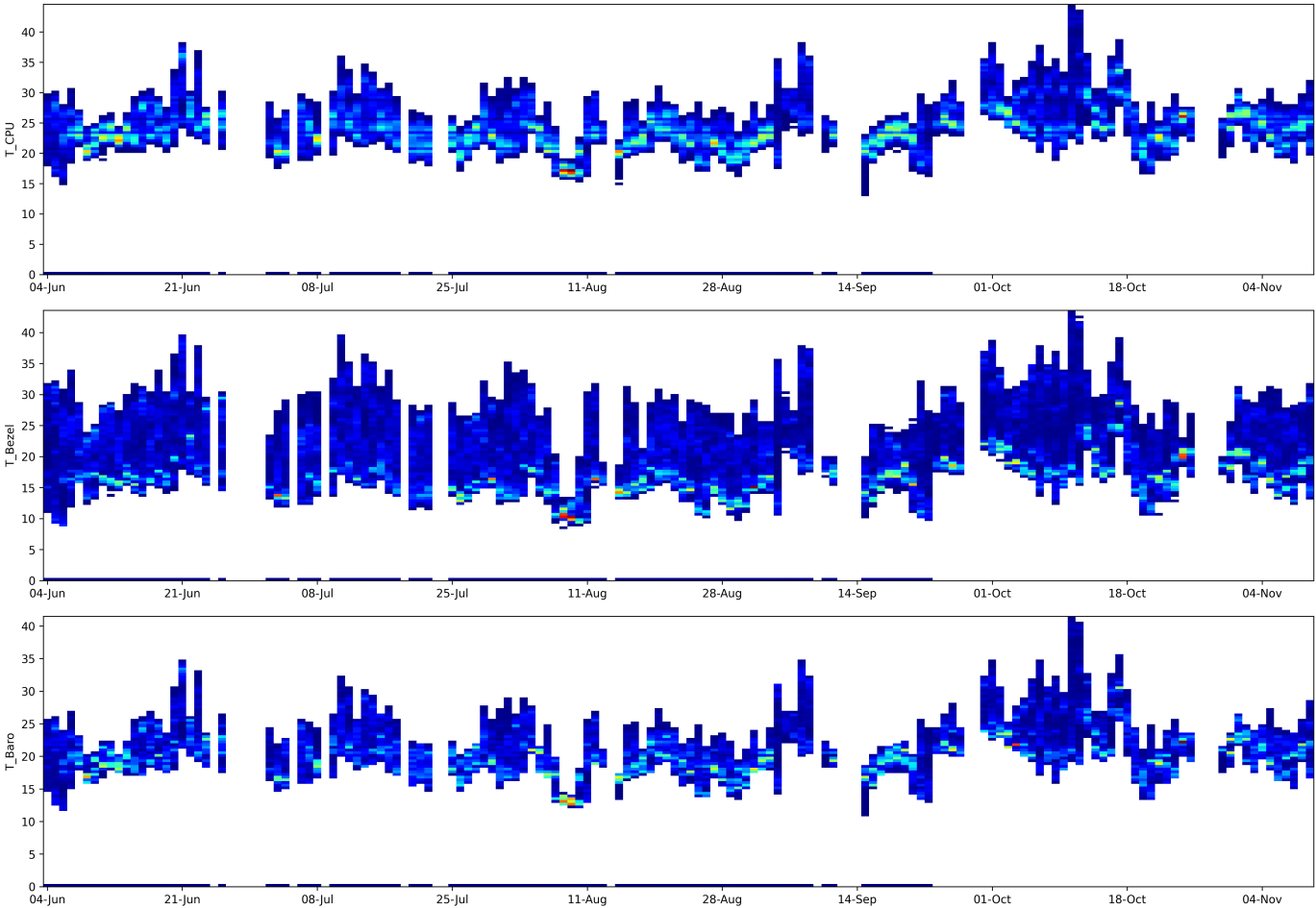
* Columns(list of strings): the names of the columns you want to plot.
* Period(int): how many days to plot per page, options: "weekly", "monthly", int
* rows(int): number of rows per page
* days\_in\_row(int): number of days per row
* flag(bool): whether or not to superimpose flagged data onto the graphs
* ignore\_zero(bool): whether or not to plot zeros
* title\_prefix(str): string that will display before the title

Note: you can’t currently plot data from the accessory\_data table, at the same time as from any other tables, as they may have different frequencies

### plot\_daily\_hists

plots a 2d histogram representing one column of data.

graph.plot\_daily\_hist(columns, title="", \*\*kwargs)



Plots a 2d histogram that covers the whole dataset. Each column shows the frequency of every value over one day, and over a whole dataset you can see the trend of the data, and also the outliers. Colour represents frequency, blue is infrequent, red is very frequent.

You can pass a list of column names and multiple plots will be displayed.

Like the daily plots, there are a few presets with columns and titles already set:

* pht\_hists() pressure, humidity and temperature
* voltage\_hists() all the internally measured voltages
* current\_hists() all the internally measured currents

parameters:

* columns(list(str)): list of column headers that will be plotted OR string of one column name.
* title(str): the title of the plot

You can also pass keyword arguments to [load](#_load_2), which will control what data is being plotted.

You can also pass keyword arguments to the underlying DailyHists function:

* ignore\_zero(bool): filters out all the zero values.
* weight(bool): weights each column of the histogram based on how many readings are in that day, means shorter days have the same impact as longer days
* bins(int): number of bins to have on the y-axis.
* zero\_axes(bool): if True, starts the y-axis from 0, rather than the lowest data value.
* ylims(int, int): can manually set the ylims.
* show\_xticks(bool): whether or not to show labels on the x-axis, useful when plotting multiple at once with the same scale.
* ybuffer(bool): if True, adds a 10% buffer to the top of the graph.
* limited\_bins(bool): if True, sets the number of bins to the number of unique readings.
* log(bool): if True, uses a log scale in the positive and negative direction.

When using a log scale(log=True), there a few parameters you can pass as keyword arguments that control how the scaling works

* linthresh, base, linscale: explained in matplotlib’s documentation [here](https://matplotlib.org/stable/api/_as_gen/matplotlib.colors.SymLogNorm.html).

These parameters control the spacing of the yticks on the log scale, and the defaults won't work for all ranges of values.

* lin\_ticks: how many ticks on the y axis to show for the linear portion, depends on how large the graph is.
* log\_ticks\_skipped: the gap between each log tick, default 0, used if the graph is small and the log ticks are overlaid on top of themselves.

### biggest\_dips

plots the wavelength of the most common peaks.

graph.biggest\_dips(n=15, cutoff\_angle=numpy.radians(80), cutoff\_wavelength=1000, date\_format="%y-%m-%d %H:%M", title=None)

A screen shot of a graph

Description automatically generated

The size of each point represents the width of the dip, the colour of each point represents the prominence.

Params:

* n(int): the number of peaks that will be selected per measurement.
* cutoff\_angle(float)(degrees): the maximum zenith angle included, used for excluding nighttime.
* cutoff\_wavelength(float): the maximum wavelength included.
* date\_format(str): the format to display the date on the x axis as. Uses [datetime.strftime](https://docs.python.org/3/library/datetime.html).
* Title(str): the title of the plot.

### daily\_biggest\_dips

makes daily plots of the location of the most prominent dips throughout each day.

graph.daily\_biggest\_dips(n=15, cutoff\_angle=80, cutoff\_wavelength=1000,

date\_format="%H:%M")

A screen shot of a computer

Description automatically generated

The size of each point represents the width of the dip, the colour of each point represents the prominence.

Params:

* n(int): the number of peaks that will be selected per measurement.
* cutoff\_angle(float)(degrees): the maximum zenith angle included, used for excluding nighttime.
* cutoff\_wavelength(float): the maximum wavelength included.
* date\_format(str): default "%M:%H" the format to display the date on the x axis as. Uses [datetime.strftime](https://docs.python.org/3/library/datetime.html).

### plot\_time­\_day

plots a variable’s change over each day in the dataset.

graph.plot\_time\_day(column, stack\_resolution="max")

A colorful image of a sound wave

Description automatically generated with medium confidence

Parameters:

* column(str): the name of the column that will be plotted.
* stack\_resolution(str): ("max", "mean", or "min"). When there are more readings in a day than pixels, how should all the readings be combined into one pixel.

### Elevation/azimuth plot

plots a graph of elevation against azimuth, with the colour representing the largest of a given value measured for each pair of elevation and azimuth values.

graph.plot\_elv\_azi(column)

A rainbow with many colors

Description automatically generated with medium confidence

currently data from accessory\_data does not work, only data from spectral\_data, system\_data, and precalculated\_values works.

params:

* column: the column that will be represented by the colourmap

### plot\_spectrum\_day

plots each spectrum over a day on a colour intensity plot

graph.plot\_spectrum\_day(normalisation=None)

A screenshot of a graph

Description automatically generated

parameters:

* normalisation: How to normalise the data. Default None
  + None: no normalisation
  + toa\_integral: divided by the top of atmosphere integral
  + pvlib: normalises against a spectrum generated by pvlib

x axis is hour of day

y axis is wavelength

colour is intensity

also plots a line graph of the global and diffuse integral, to get an idea of what the cloud coverage was like at each point in the day.

### plot\_aod\_day

graph.plot\_aod\_day(clearsky\_filter="wood", clearsky\_filter\_kwargs={})

Plots the Aerosol Optical Depth across the spectrum, one page per day

A screenshot of a computer screen

Description automatically generated

* parameters:
  + clearsky\_filter: which clearsky filtering method to use. currently only "wood" is implemented. if None, no filtering is applied
  + clearsky\_filter\_kwargs: keyword arguments to pass to the clearsky\_filter method, these will vary depending on which filtering method is used. The keyword arguments for each method are listed below.

x axis is hour of day

y axis is wavelength

colour is the optical depth at that wavelength and time

summary of clearsky filtering options:

"wood" – currently the only implemented method, keeps all readings where the aod at 500nm is less than 2, and is within 0.05 of all readings within 4 minutes.

Parameters(to be passed into clearsky\_filter\_kwargs):

* reading\_index: the wavelength that is used for the clearsky calculations. if on a full 300-1100nm spectra, make sure to subtract 300 from the wavelength you want to use
* absolute\_filter: any readings higher than this will be filtered out
* relative\_filter: any readings that are more than this value more or less than any other within the time window will be filtered out
* relative\_time\_period: the time period over which the relative filtering is done. this is the total time period that the filtering is done by, so if you want 5 mins either side, pass 10min. format is a string that will be passed to pd.Timedelta. documentation: https://pandas.pydata.org/docs/reference/api/pandas.Timedelta.html

also plots a line graph of the global integral, diffuse integral, and total optical depth at 500nmx500 to get an idea of what the cloud coverage was like at each point in the day.

total\_od is the total optical depth, with no corrections

aod\_microtops is the aerosol optical depth, with the Rayleigh scattering removed

aod\_wood\_2017 is the same as the above value but with an empirical correction as detailed in: [Wood, J., Smyth, T. J., and Estellés, V.: Autonomous marine hyperspectral radiometers for determining solar irradiances and aerosol optical properties, Atmos. Meas. Tech., 10, 1723–1737, https://doi.org/10.5194/amt-10-1723-2017, 2017.](https://amt.copernicus.org/articles/10/1723/2017/)

**daily\_aod\_cimel**

Plots several line graphs of the AOD sampled at certain passed wavelengths, defaults to the wavelengths measured by Cimel spectrometers: [380, 440, 500, 675, 870, 1020]

graph.daily\_aod\_cimel(aod\_type="aod\_microtops", wavelengths=None,

period=21, rows=3, days\_in\_row=7,

upper\_limit=2, lower\_limit=0,

clearsky\_filter="wood", clearsky\_filter\_kwargs={})

A white screen with black text

Description automatically generated

Parameters:

* aod\_type: one of: total\_od, aod\_microtops, aod\_wood\_2017. which aod calculation to use.
* period(int or string): how many days to plot per page, options: "weekly", "monthly", int.
* rows: number of rows per page
* days\_in\_row: number of days per row
* upper\_limit: the max value to display
* lower\_limit: the minimum value to display
* clearsky\_filter: which clearsky filter algorithm to use
* clearsky\_filter\_kwargs: keyword arguments to pass to clearsky\_filter

summary of clearsky filtering options:

"wood" – currently the only implemented method, keeps all readings where the aod at 500nm is less than 2, and is within 0.05 of all readings within 4 minutes.

Parameters(to be passed into clearsky\_filter\_kwargs):

* reading\_index: the wavelength that is used for the clearsky calculations. if on a full 300-1100nm spectra, make sure to subtract 300 from the wavelength you want to use
* absolute\_filter: any readings higher than this will be filtered out
* relative\_filter: any readings that are more than this value more or less than any other within the time window will be filtered out
* relative\_time\_period: the time period over which the relative filtering is done. this is the total time period that the filtering is done by, so if you want 5 mins either side, pass 10min. format is a string that will be passed to pd.Timedelta. documentation: <https://pandas.pydata.org/docs/reference/api/pandas.Timedelta.html>

### wait\_until\_closed

graph.wait\_until\_closed()

stops matplotlib from closing all the plots when the last one is generated. only relevant when block=False in Graph.

## read\_txt

the read\_txt module is used to read the .txt files that are outputted by older hsr1 modules and convert it into a format that can be stored in the database. there are three functions you might use, read(), which reads the most commonly used data, and the global and diffuse spectra. read\_raw\_txt() reads each channel’s spectral readings. read\_raw\_pixels() reads each channel’s raw pixel data, before it is converted into spectral data

functions:

* read
* read\_raw\_txt
* read\_raw\_pixels

### read

read(hsr\_path="", start\_date="2000-01-01", end\_date="2100-01-01", deployment\_metadata\_filepath=None)

reads all the commonly used data from the .txt files. only loads the global and diffuse spectrum

* params:
  + hsr\_path: the filepath to the data that will be read. this folder should contain one .zip file for each day, named YYYY-MM-DD.zip
  + start\_date: the first date that is read from the folder of zip files, inclusive. should be in the format YYYY-MM-DD
  + start\_date: the last date that is read from the folder of zip files, inclusive. should be in the format YYYY-MM-DD
  + deployment\_metadata\_filepath: the filepath to the deployment metadata .ini file that will be read.
* returns:
  + tuple of dataframes that can be stored to the database using DBDriver.store(). one dataframe per table in the database

### read\_raw\_txt

read\_raw\_txt(hsr\_path="", start\_date="2000-01-01",

end\_date="2100-01-01", deployment\_metadata\_filepath=None)

reads each channel of spectral data from the .txt files

* params:
  + hsr\_path: the filepath to the data that will be read. This folder should contain one .zip file for each day, named YYYY-MM-DD.zip
  + start\_date: the first date that is read from the folder of zip files, inclusive. Should be in the format YYYY-MM-DD
  + start\_date: the last date that is read from the folder of zip files, inclusive. Should be in the format YYYY-MM-DD
  + deployment\_metadata\_filepath: the filepath to the deployment metadata .ini file that will be read
* returns a tuple of dataframes that can be used with rawDBDriver.store to make a raw dataset

### read\_raw\_pixels

reads the raw pixel data for each channel in the .txt files

* params:
  + file\_path: the filepath to the data that will be read. This folder should contain one .zip file for each day, named YYYY-MM-DD.zip
  + skiprows: the nuber of rows at the top of each file to skip when reading

## utils.HsrFunc

utils.hsrfunc is a module that has a variety of functions that calculate derived values from hsr1’s output data

functions:

* calc\_direct\_normal\_spectrum
* calc\_sun\_zenith
* calc\_rayleigh
* calc\_air\_mass
* calc\_aot\_direct
* calc\_cimel\_band\_aot\_direct
* calc\_aod\_from\_df
* load\_et\_spectrum
* calculate\_clearsky\_filter

### calc\_direct\_normal\_spectrum

hsr1.utils.HsrFunc.calc\_direct\_normal\_spectrum(global\_spectrum,

diffuse\_spectrum, sza

calculates the direct normal spectrum from the diffuse, global and zenith angle

params:

* global\_spectrum, diffuse\_spectrum: either pandas Series of numpy arrays, each array representing one spectral reading or dataframe with one row per wavelength reading.
* sza: pandas Series one float(radians) per reading

returns the direct normal spectrum, in the same format as the spectral inputs

### calc\_sun\_zenith

hsr1.utils.HsrFunc.calc\_direct\_normal\_spectrum(global\_spectrum,

diffuse\_spectrum, sza

calculates the solar zenith angle at a specific time and location

quite slow as this has to be run in a for loop, sg2's calculation is much faster.

params:

* ts: timestamp that you want to measure, string yyyy-mm-dd hh:mm:ss
* lat, lon: latitude and longitude of the location

returns:

* tuple of zenith angle and azimuth, in radians

### calc\_rayleigh

hsr1.utils.HsrFunc.calc\_rayleigh(wl, pressure=1013.25)

calculates the rayleigh scattering

params:

* wl: the wavelength(s) to calculate the scattering over. This can be a scalar, numpy array or pandas series
* pressure: the pressure in hPa

return has the same datatype as the wl parameter.

### calc\_air\_mass

hsr1.utils.HsrFunc.calc\_air\_mass(Sza, pressure=1013.25)

calculates the airmass

params:

* Sza: the solar zenith angle in radians. This can be a scalar, numpy array or pandas series.
* pressure: the pressure in hPa

return is in the same datatype as the wl parameter

### calc\_aot\_direct

hsr1.utils.HsrFunc.calc\_aot\_direct(ed, eds, sza, e\_solar=None, sed=None,

aod\_type=["total\_od", "aod\_microtops",

"aod\_wood\_2017"], et\_wavelengths=None)

calculates the atmospheric optical depth in several different ways

params:

* ed: the global spectral irradiance. columns=wavelength, index=time, timestamp in utc
* eds: the diffuse spectral irradiance. columns=wavelength, index=time, timestamp in utc
* sza: the solar zenith angle in radians. column = "sza"
* e\_solar: the extraterrestrial solar spectrum. columns=wavelength, index=time, timestamp in utc
* sed: sun earth distance in AU, pandas Series. if None, calculated from time. this is quite slow so this parameter can speed it up if you are already loading from the database
* aod\_type: the desired outputs string or list of strings

returns:

* aod\_data: dataframe containing all the requested channels

to convert a column of numpy arrays into a dataframe of wavelengths against time:

pd.DataFrame(np.stack(data["spectrum\_column"].values),

columns=np.arange(300, 1101, 1), index=data["pc\_time\_end\_measurement"])

definition of each aod type:

* tau\_t/total\_od: the total optical depth from all sources
* tau\_a/microtops\_aod: the aerosol optical depth. this is the optical depth with rayleigh scattering removed
* tau\_corr/aod\_wood\_2017: the aerosol optical depth with an empirical correction applied to it

hsr1.utils.HsrFunc.calc\_cimel\_band\_aot\_direct(Ed, Eds, Sza, E\_solar, sed,

aod\_type=["total\_od",

"aod\_microtops",

"aod\_wood\_2017"])

### calc\_cimel\_band\_aot\_direct

calculates the AOD values at several wavelengths, that are also measured by cimel instruments

params:

* ed: the global spectral irradiance. columns=wavelength, index=time, timestamp in UTC
* eds: the diffuse spectral irradiance. columns=wavelength, index=time, timestamp in UTC
* sza: the solar zenith angle in radians. column = "sza", index=time, timestamp in UTC
* e\_solar: the extraterrestrial solar spectrum. columns=wavelength, index=time, timestamp in UTC
* sed: sun earth distance in AU. if None, calculated from time. this is quite slow so this parameter can speed it up if you are already loading from the database
* aod\_type: the desired outputs string or list of strings

returns:

* aod\_data: dataframe containing all the requested channels

to convert a column of numpy arrays into a dataframe of wavelengths against time:

pd.DataFrame(np.stack(data["spectrum\_column"].values),

columns=np.arange(300, 1101, 1), index=data["pc\_time\_end\_measurement"])

definition of each aod type:

* tau\_t/total\_od: the total optical depth from all sources
* tau\_a/microtops\_aod: the aerosol optical depth. this is the optical depth with rayleigh scattering removed
* tau\_corr/aod\_wood\_2017: the aerosol optical depth with an empirical correction applied to it

### calc\_aod\_from\_df

hsr1.utils.HsrFunc.calc\_aod\_from\_df(data, cimel=False,

aod\_type=["total\_od",

"aod\_microtops", "aod\_wood\_2017"],

wavelengths=None)

coerces data from the format returned from the database to the format for the calculation, and back, and loads the reference solar spectrum

params:

* data: dataframe of all the relevant data,
  + columns = ["pc\_time\_end\_measurement", "global\_spectrum", "diffuse\_spectrum", "sza", "sed"]
* cimel: if True, calculates the aod only at certain wavelengths that cimel instruments measure at
* aod\_type: the desired outputs string or list of strings
* wavelengths: filters the wavlengths that are used for the caluclation, and that are returned

returns:

* aod\_data: dataframe containing all the requested channels

to convert a column of numpy arrays into a dataframe of wavelengths against time:

pd.DataFrame(np.stack(data["spectrum\_column"].values),

columns=np.arange(300, 1101, 1), index=data["pc\_time\_end\_measurement"])

definition of each aod type:

* tau\_t/total\_od: the total optical depth from all sources
* tau\_a/microtops\_aod: the aerosol optical depth. this is the optical depth with rayleigh scattering removed
* tau\_corr/aod\_wood\_2017: the aerosol optical depth with an empirical correction applied to it

### load\_et\_spectrum

hsr1.utils.HsrFunc.load\_et\_spectrum(filepath=None, wavelengths=None)

loads a .txt file containing a reference extraterrestrial solar spectrum, and applies some smoothing

params:

* filepath: filepath to the file where the spectrum is stored. if None, a default spectrum that comes with the library is used
* wavelengths: which wavelengths to use from the reference file

returns a dataframe with one row which contains the reference spectrum

### calculate\_clearsky\_filter

hsr1.utils.HsrFunc.calculate\_clearsky\_filter(data:pd.DataFrame,

global\_spectrum=None,

diffuse\_spectrum=None,

method="wood", kwargs={})

method that calls the appropriate clearsky function

params:

* data: the data that is used for the clearsky calculation
* method: the method that will be used to calculate which readings are clearsky readings
* kwargs: the keyword arguments to pass to clearsky\_filter

returns a numpy array the same length as input dataframe with ones and zeros, 1=clearsky 0=cloud

# Database reference

Data from an HSR1 is stored across several tables in a sqlite database.

## spectral\_data

this is where the sunlight data is stored. This has the global and diffuse integral and spectral data.

Columns:

* pc\_time\_end\_measurement(string): the timestamp of the reading. In local time with a timezone. This time is the time at the end of the period where the readings were averaged and stored. For example, if readings are taken every 10 seconds and averaged every minute, the readings at 12:00:10, 12:00:20, 12:00:30, 12:00:40, 12:00:50, and 12:01:00 would all be stored under the timestamp 12:01:00.
* sample\_id(string): a UUID that is unique to this measurement. The same sample ID is used for the measurement’s system\_data, precalculated\_values and the accessory\_data that was generated at the same timestamp.
* dataseries\_id(string): a UUID that is unique to each dataseries. A Dataseries is a run of data collection that has all the same settings. If a run of data is stored incrementally to the database, each section is stored as a separate dataseries. Each dataseries id is shared between all the tables of the database.
* global\_spectrum(bytes): the global spectral output of the instrument. The data is compressed to save storage space and time, but when loading from the database it is automatically decompressed into a numpy array of floats for each measurement. The data is encoded from a 1d numpy array using numpy’s [array.tobytes() function](https://numpy.org/doc/stable/reference/generated/numpy.ndarray.tobytes.html).
* diffuse\_spectrum(bytes): the diffuse spectral output of the instrument. The data is compressed to save storage space and time, but when loading from the database it is automatically decompressed into a numpy array of floats for each measurement. The data is decoded using numpy’s [array.frombuffer()](https://numpy.org/doc/stable/reference/generated/numpy.frombuffer.html#numpy.frombuffer) function
* global\_integral(float): global solar intensity. The integrated value of the measured spectrum over 400nm – 1000nm.
* diffuse\_integral(float): diffuse solar intensity. The integrated value of the measured spectrum over 400nm – 1000nm.
* global\_molar(float): global molar intensity. The integrated value of the measured spectrum over 400nm – 700nm with a quantum weighting.
* diffuse\_molar(float): The integrated value of the measured spectrum over 400nm – 700nm with a quantum weighting.
* camera\_temp(float): the temperature measured in the camera. A few degrees higher than ambient.

## system\_data

data from the internal GPS module. If the instrument has accessory\_data, columns from accessory\_data are just renamed and averaged from that.

Averaging specifics:

Each reading in spectral\_data has a corresponding reading in system\_data. All readings in accessory\_data are averaged to the next reading in spectral\_data. In cases where there has been a dropout of spectral\_data, the first system\_data reading once back is deleted, as if the accessory\_data file is still running, then all the data that was taken while there was no spectral\_data readings would all be averaged in to one measurement, which would be misleading.

Columns:

* pc\_time\_end\_measurement(string): the timestamp of the reading. In local time with a timezone. This time is the time at the end of the period where the readings were averaged and stored. For example, if readings are taken every 10 seconds and averaged every minute, the readings at 12:00:10, 12:00:20, 12:00:30, 12:00:40, 12:00:50, and 12:01:00 would all be stored under the timestamp 12:01:00.
* gps\_time(string): the time as measured by the GPS module in UTC. The difference between this and the pc time can be useful for finding when the GPS is inaccurate.
* gps\_status(string): the status of the GPS. A=Valid, V=Invalid.
* gps\_latitude(float): the latitude in degrees.
* gps\_longitude(float): the longitude in degrees.
* gps\_altitude(float): the height above sea level, in meters
* pressure(float): the pressure in hPa.
* baro\_temp(float): the temperature measured by the barometer in Celsius.
* rh(float): the relative humidity. Measured as the percentage of water in the air, where condensation starts at 100% rh.
* rh\_temp(float): the temperature measured by the rh sensor in Celsius.
* dataseries\_id(string): a UUID that is unique to each dataseries. A Dataseries is a run of data collection that has all the same settings. If a run of data is stored incrementally to the database, each section is stored as a separate dataseries. Each dataseries id is shared between all the tables of the database.
* sample\_id(string): a UUID that is unique to this measurement. The same sample ID is used for the measurement’s system\_data, precalculated\_values and the accessory\_data that was generated at the same timestamp.

## precalculated\_values

a few columns of values that are commonly used in calculations. Having these values in the database means they’re only calculated once, saving processing time when using the data.

Columns:

* pc\_time\_end\_measurement(string): the timestamp of the reading. In local time with a timezone. This time is the time at the end of the period where the readings were averaged and stored. For example, if readings are taken every 10 seconds and averaged every minute, the readings at 12:00:10, 12:00:20, 12:00:30, 12:00:40, 12:00:50, and 12:01:00 would all be stored under the timestamp 12:01:00.
* sza(float): the solar zenith angle, in radians. This is the angle between the sun and the zenith (the point in the sky directly upwards).
* azimuth(float): the solar azimuth angle, in radians. This is the horizontal angle between the sun and true north.
* toa\_hi(float): w/m^2. the horizontal solar intensity at the top of the atmosphere. To calculate the direct normal intensity, multiply this by cos(sza).
* Sed(float): the distance between the sun and the earth in AU. This is caused by the earth’s orbit being elliptical, not circular.
* Airmass(float): the optical path length. Expressed as a ratio to the minimum path length (vertically upwards).
* sample\_id(string): a UUID that is unique to this measurement. The same sample ID is used for the measurement’s system\_data, precalculated\_values and the accessory\_data that was generated at the same timestamp.
* dataseries\_id(string): a UUID that is unique to each dataseries. A Dataseries is a run of data collection that has all the same settings. If a run of data is stored incrementally to the database, each section is stored as a separate dataseries. Each dataseries id is shared between all the tables of the database.

## deployment\_metadata

all the metadata stored in the deployment metadata .ini file. Contains data about the deployment and also the specific dataseries. Each row corresponds to one dataseries, and a new row is added each time data is appended to the database.

Columns:

* aux\_average\_period(int): the time period over which the GPS readings are averaged. This is the period of all the measurements in the system\_data table. If the device measures accessory\_data, the period will be the same as the spectral measurements.
* aux\_sampling\_period(int): the time period over which readings are taken. Readings are then averaged according to aux\_average\_period. If the device measures accessory\_data, that will be recorded with the same period as the spectrometer
* calibration\_comment(string): comment to describe where and how the calibration was made.
* calibration\_time(string): timestamp when the instrument was calibrated.
* camera\_calibration\_file(string): name of the calibration file containing the camera’s calibration.
* camera\_id(string): the camera’s id (serial number).
* data\_structure\_id(string): the version of the database
* dataseries\_id(string): a UUID that is unique to each dataseries. A dataseries is a run of data collection that has all the same settings. If a run of data is stored incrementally to the database, each section is stored as a separate dataseries. Each dataseries id is shared between all the tables of the database.
* default\_elevation(float): the elevation value to use when the GPS is inactive.
* default\_latitude(float): the latitude value to use when the GPS is inactive.
* default\_longitude(float): the longitude value to use when the GPS is inactive.
* deployment\_description(string): a short description of the current deployment.
* deployment\_id(string): a UUID that is shared across all the dataseries in the deployment. New data will automatically share existing deployment\_ids if they have the same values in all of the columns.
* gain(float): the gain setting for the spectrometer.
* hdr(float): True if the spectrometer is measuring in high dynamic range mode.
* integration\_time(float): the integration time (ms) for the spectrometer.
* license(string): the name of the license that the data is held under.
* license\_reference(string): web link to the above license.
* location\_name(string): the name of the location where this dataseries is measured from.
* mobile(int): is the dataseries mobile or not 0=static, 1=mobile.
* operator\_contact(string): contact details for the operator of this instrument
* owner\_contact(string): contact details for the owner of this instrument.
* platform\_id(string): reference to platform or campaign metadata if known.
* processing\_level(int): 0 - raw detector values; 1 - calibrated detector output e.g. my Raw files; 2 - recalibrated, or derived data e.g. Global & Diffuse; 3 - further interpreted or adjusted
* sensor\_id(string): the id (label serial nimber) of this instrument.
* sensor\_type(string): the type of this instrument.
* software\_id(string): the version of the software currently running on the instrument.
* spectrometer\_average\_period(float): the time period over which the spectrometer readings are averaged.
* spectrometer\_burst\_number(float): number of consecutive readings in each sample.
* spectrometer\_calibration\_file(string): name of the file containing the spectrometer calibration.
* spectrometer\_sampling\_period(float): the period between each spectrometer reading.
* start\_time(string): timestamp of the start of the dataseries.
* timezone(string): the local timezone at the time of recording in the form "+/-hh:mm". timestamps are stored as local time with this timestamp, so if you want to account for daylight-saving time, you should start a new dataseries when the clocks change with the new timezone.
* wavelengths(string): the start, stop, and step of the measured wavelengths.

## accessory\_data

this table contains the contents of the accessory\_data.txt file. It is more detailed than GPS.txt, but only exists on more modern instruments.

* pc\_time\_end\_measurement(string): the timestamp of the reading. In local time with a timezone. This time is the time at the end of the period where the readings were taken. For example, if readings are taken every 10 seconds, the readings at 12:00:10, 12:00:20, 12:00:30, 12:00:40, 12:00:50, and 12:01:00 would all be stored under the timestamp 12:01:00.
* Clock\_error(float): difference between the control PCB CPU clock and GPS time.
* Latitude(float): the Latitude measured by the GPS
* Longitude(float): the Longitude measured by the GPS
* Altitude(float): the Altitude measured by the GPS
* GPSAge(float): time since last valid GPS reading, in seconds
* NSV(float): Number of satellites in view, across all constellations
* GPSSpare(float): Not in use
* \_15Vin(float): Input voltage
* \_3VCAP(float): Voltage at storage capacitor for internal PCB processor
* HTRIn(float): Voltage at heater power input
* Vcc(float): Control PCB processor supply rail
* VSpare(float): Not in use
* I\_Tot(float): Total input current (mA)
* I\_15VPC(float): PC current (mA) (high voltage supply)
* I\_15VCAM(float): Cameral current (mA)
* I\_5VPC(float): PC current – 5V supply
* ISpare(float): Not in use
* T\_CPU(float): Temperature measured at the CPU, in Celsius.
* T\_Bezel(float): Temperature measured at the bezel (optical head) in °C
* T\_RH(float): Temperature measured by the relative humidity sensor, in Celsius.
* T\_Baro(float): Temperature measured by the barometer, in Celsius.
* RH(float): the relative humidity. Measured as the percentage of water in the air, where condensation starts at 100% rh.
* Pressure(float): the pressure in hPa.
* IMU\_Output\_Type(float): Type number for inertial output values
* Yaw(float): the Yaw, in degrees.
* Roll(float): the Roll, in degrees.
* Pitch(float): the Pitch, in degrees.
* Q\_W(float): values representing the orientation in quaternions.
* Q\_X(float): values representing the orientation in quaternions.
* Q\_Y(float): values representing the orientation in quaternions.
* Q\_Z(float): values representing the orientation in quaternions.
* Acc\_X(float): Acceleration in the X direction
* Acc\_Y(float): Acceleration in the Y direction
* Acc\_Z(float): Acceleration in the Z direction
* IMU1(float): Not in use
* IMU2(float): Not in use
* Control\_flags(float): Copy of the internal control flags
* Control\_flags2(float): Copy of the internal control flags
* StatusFlags(float): Copy of internal status flags
* Heater(float): Heater control PWM value
* MsgCount(float): Count of number of Modbus messages received correctly
* RdgCount(float): Count of number of readings incorporated in this reading value
* Watchdog(float): Count of watchdog – PC is reset if this gets to zero
* gps\_time(string): a timestamp containing the time as measured by the GPS unit. When it is working correctly, it should be the same as pc\_time\_end\_measurement
* cpu\_time(string): a timestamp containing the time measured by the CPU
* dataseries\_id(string): a UUID that is unique to each dataseries. A Dataseries is a run of data collection that has all the same settings. If a run of data is stored incrementally to the database, each section is stored as a separate dataseries. Each dataseries id is shared between all the tables of the database.

# raw\_data

this table contains the raw data, before the global and diffuse split is calculated. Each reading is a spectrum of wavelength intensities, from 300-1100nm, like the global and diffuse spectra above. There is one column in this table for each channel that measurements are made from, the number of which varies between different versions of the hsr1 instrument. The spectra are serialised, for more efficient storage and faster loading, but the serialisation and deserialization process is all handled by the library’s load and store functions.

## Deployment.ini file

This file provides deployment metadata for the conversion of text files to the database. It is stored as a text file, and can have any valid name.

The example below shows a typical file format. This should be edited to match your specific dataset, and is used by the db\_driver.store() function.

[deployment]

# Information relating to the overall campaign

content = deployment\_metadata

data\_structure\_id = 1.0

deployment\_id =

deployment\_description = HSR1-005 at Winster 2024

sensor\_type = HSR1 prototype v2 general reference

sensor\_id = HSR1-005

camera\_id = 700007631242

owner\_contact = john@peakdesign.co.uk

operator\_contact = John Wood

license = CC BY-NC 4.0

license\_reference = https://creativecommons.org/licenses/by-nc/4.0

[dataseries]

# information relating to this particular logging series

processing\_level = 2

# timezone of PC clock

timezone = 00:00

# set these values for a stationary setup, or if the GPS is not fitted

default\_latitude = 53.14

default\_longitude = -1.63

default\_elevation = 240

#This information is taken from the logging software or datafiles

software\_id = 1.0.0.11

calibration\_time = 2024-02-01

calibration\_comment = Calibration against HSR1-001 & HSR1-003 at Winster

camera\_calibration\_file = 'Baumer 700007631242 CameraCalibration 20ms 4seq 2024-02-01.txt'

spectrometer\_calibration\_file = 'Baumer 700007631242 SpectrometerCalibration 2024-02-02 C.tx'

start\_time = ''

integration\_time = 5

gain = 2

hdr = 1

spectrometer\_sampling\_period = 10

spectrometer\_average\_period = 60

spectrometer\_burst\_number = 1

wavelengths = 300, 1100, 1

aux\_sampling\_period = 10

aux\_average\_period = 60

## License

This program is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.