oucass-profiles Documentation

Release 1.3.0

January 28, 2021

Table of Contents

Coef_Manager

class Coef_Manager.**Azure_Coef_Manager** (*table_service*)

interface with Azure

get_coefs (*type*, *serial_number*)

Get the coefs for the sensor with the given type and serial number.

Parameters
- **type** (*str*) – “Imet” or “RH” or “Wind”
- **serial_number** (*str*) – the sensor’s serial number

Return type dict

Returns information about the sensor, including offset OR coefs and calibration equation

get_sensors (*scoopID*)

Get the sensor serial numbers for the given scoop.

Parameters **scoopID** (*str*) – The scoop’s identifier

Return type dict

Returns sensor numbers as {“imet1”:””, “imet2”:””, “imet3”:””, “imet4”:””, “rh1”:””, “rh2”:””, “rh3”:””, “rh4”:””}

get_tail_n (*copterID*)

Get the tail number corresponding to a short ID number.

Parameters **copterID** (*int*) – the short ID number of the copter

Return type str

Returns the tail number

class Coef_Manager.**CSV_Coef_Manager** (*file_path*)

get_coefs (*type*, *serial_number*)

Get the coefs for the sensor with the given type and serial number.

Parameters
- **type** (*str*) – “Imet” or “RH” or “Wind”
- **or int] serial_number** (*[str*) – the sensor’s serial number

Return type dict

Returns information about the sensor, including offset OR coefs and calibration equation

get_sensors (*scoopID*)
Get the sensor serial numbers for the given scoop.

Parameters **scoopID** (*str*) – The scoop's identifier

Return type dict

Returns sensor numbers as {"imet1":"", "imet2":"", "imet3":"", "imet4":"", "rh1":"", "rh2":"", "rh3":"", "rh4":""}

get_tail_n (*copterID*)
Get the tail number corresponding to a short ID number.

Parameters **copterID** (*int*) – the short ID number of the copter

Return type str

Returns the tail number

class Coef_Manager.**Coef_Manager**
Reads the .profilesrc file to determine if the coefs are in the local file system or on Azure and to determine the file path or connection string, then ingests the data from the proper source. This object can then be queried by scoop number (to get sensor numbers), by sensor numbers (to get coefs), or by copter number (to get tail number).

get_coefs (*type*, *serial_number*)
Get the coefs for the sensor with the given type and serial number.

Parameters
- **type** (*str*) – "Imet" or "RH" or "Wind"
- **or int] serial_number** (*[str*) – the sensor's serial number

Return type dict

Returns information about the sensor, including offset OR coefs and calibration equation

get_sensors (*scoopID*)
Get the sensor serial numbers for the given scoop.

Parameters **scoopID** (*str*) – The scoop's identifier

Return type dict

Returns sensor numbers as {"imet1":"", "imet2":"", "imet3":"", "imet4":"", "rh1":"", "rh2":"", "rh3":"", "rh4":""}

get_tail_n (*copterID*)
Get the tail number corresponding to a short ID number.

Parameters **copterID** (*int*) – the short ID number of the copter

Return type str

Returns the tail number

class Coef_Manager.**Coef_Manager_Base**

get_coefs (*type*, *serial_number*)
Get the coefs for the sensor with the given type and serial number.

Parameters
- **type** (*str*) – "Imet" or "RH" or "Wind"
- **serial_number** (*str*) – the sensor's serial number

Return type dict

Returns information about the sensor, including offset OR coefs and calibration equation

get_sensors (*scoopID*)

Get the sensor serial numbers for the given scoop.

Parameters **scoopID** (*str*) – The scoop's identifier

Return type dict

Returns sensor numbers as {"imet1":"", "imet2":"", "imet3":"", "imet4":"", "rh1":"", "rh2":"", "rh3":"", "rh4":""}

get_tail_n (*copterID*)

Get the tail number corresponding to a short ID number.

Parameters **copterID** (*int*) – the short ID number of the copter

Return type str

Returns the tail number

Meta

class Meta.**Meta** (*header_path=None*, *flight_path=None*)

Processes, stores, and writes metadata files (JSON-LD for public, CSV for private) for a flight

var dict <str; Object> all_fields

Dictionary containing all information to be written to metadata files

var list <str> private_fields

List of fields to be included in the CSV file

var list <str> public_fields

List of fields to be included in the JSON file

__init__ (*header_path=None*, *flight_path=None*)

Creates new object of type Meta from header file and flight file. Details about the input files can be found at the bottom of this page.

Parameters

- **header_path** (*str*) – path to the header file
- **flight_path** (*str*) – path to the flight file

combine (*other*)

Merge two Meta objects to create a file that accurately describes ALL related header and flight files. Only fields that are the same for both Meta objects are included; all others are set to None.

Parameters **other** (*Meta*) – the Meta object to merge into this one. Only this object (self) will be altered

get (*name*)

Request the value of a metadata field

Parameters **name** (*str*) – the name of the field

Returns the value of the field, if found

read_file (*csv_path*)

Copy data from the CSV file to the all_fields dictionary

Parameters **csv_path** (*str*) – path to the CSV data file

write_public_meta (*out_path*, *include_private=False*)

Write a human-readable text file containing metadata for the

flight. Unless include_private is set to True, only fields specified in public_fields will be included in this metadata file.

Parameters

- **out_path** (*str*) – where to save the file
- **include_private** (*bool*) – Specify True to include information intended for internal use only in the new metadata file. Default False.

Chapter 3

Profile

Manages data from a single flight or profile

class Profile.**Profile** (**args*, ***kwargs*)

A Profile object contains data from a profile (if altitude or pressure is specified under resolution) or flight (if the resolution is in units of time)

Variables

- **dev** (*bool*) – True if data is from developmental flights
- **resolution** (*Quantity*) – resolution of the data in units of altitude or pressure
- **indices** (*tuple*) – the bounds of the profile to be processed as (start_time, end_time)
- **ascent** (*bool*) – True if data from the ascending leg should be processed, otherwise the descending leg will be processed instead
- **file_path** (*String*) – the path to you .bin, .json, or .nc data file
- **gridded_times** (*np.Array<Datetime>*) – the times at which data points are generated
- **gridded_base** (*np.Array<Quantity>*) – the value of the vertical coordinate at each data point

get (*varname*)

Returns the requested variable, which may be in Profile or one of its attributes (ex. temp is in thermo_profile)

Parameters **varname** (*str*) – the name of the requested variable

Returns the requested variable

get_thermo_profile ( )

If a Thermo_Profile object does not already exist, it is created when this method is called.

Returns the Thermo_Profile object

Return type *Thermo_Profile*

get_wind_profile ( )

If a Wind_Profile object does not already exist, it is created when this method is called.

Returns the Wind_Profile object

Return type *Wind_Profile*

Chapter 4

Profile_Set

Manages data from a collection of flights or profiles at a specific location

class Profile_Set.**Profile_Set** (*resolution=10*, *res_units='m'*, *ascent=True*, *dev=False*, *confirm_bounds=True*, *profile_start_height=None*, *nc_level='none'*)

This class manages data (in the form of Profile objects) from one or many flights.

Variables

- **profiles** (*list<Profile>*) – list of Profile objects at this location
- **ascent** (*bool*) – True if data from the ascending leg of the profile is to be used. If False, the descending leg will be processed instead
- **dev** (*bool*) – True if data from developmental flights is to be uploaded
- **resolution** (*int*) – the vertical resolution desired
- **res_units** (*str*) – the units in which the vertical resolution is given
- **confirm_bounds** (*bool*) – if True, the user will be asked to verify the automatically-determined start, peak, and end times of each profile
- **profile_start_height** (*int*) – either passed to the constructor or provided by the user during processing
- **meta** (*Meta*) – reads and processes metadata from oucass-checklist

__init__ (*resolution=10*, *res_units='m'*, *ascent=True*, *dev=False*, *confirm_bounds=True*, *profile_start_height=None*, *nc_level='none'*)

Creates a Profiles object.

Parameters

- **resolution** (*int*) – resolution to which data should be calculated in units of altitude or pressure
- **res_units** (*str*) – units of resolution in a format which can be parsed by pint
- **ascent** (*bool*) – True to use ascending leg of flight, False to use descending leg
- **dev** (*bool*) – True if data is from a developmental flight
- **confirm_bounds** – False to bypass user confirmation of automatically identified start, peak, and end times
- **profile_start_height** (*int*) – if provided, the user will not be prompted to enter the starting height for each profile separately. This can be usefull when processing many profiles from the same mission, but at least one profile should be processed without this parameter to deter-

mine its correct value.

- **nc_level** (*str*) – either 'low', or 'none'. This parameter is used when processing non-NetCDF files to determine which types of NetCDF files will be generated. For individual files for each Raw, Thermo, and Wind Profile, specify 'low'. For no NetCDF files, specify 'none'. To generate a single, Profile_Set-level file, call Profile_Set.save_netCDF where you are done adding data.

__str__ ( )
Return str(self).

add_all_profiles (*file_path*, *scoop_id=None*, *meta_flight_path=None*, *meta_header_path=None*)
Reads a file, splits it in to several vertical profiles, and adds all Profiles to profiles

Parameters
- **file_path** (*str*) – the data file
- **scoop_id** (*str*) – the identifier of the sensor package used
- **meta_flight_path** (*str*) – path to the "flight" file generated by oucass-checklist
- **meta_header_path** (*str*) – path to the "header" file generated by oucass-checklist

add_profile (*file_path*, *time=datetime.datetime(1, 1, 1, 0, 0)*, *profile_num=None*, *scoop_id=None*, *meta_header_path=None*, *meta_flight_path=None*)
Reads a file and creates a Profile for the first vertical profile after time OR for the profile_numth profile.

Parameters
- **file_path** (*string*) – the data file
- **time** (*datetime*) – the time after which the profile begins (used only if profile_num is not specified)
- **profile_num** (*int*) – use the nth profile in the file
- **scoop_id** (*str*) – the identifier of the sensor package used
- **meta_flight_path** (*str*) – path to the "flight" file generated by oucass-checklist
- **meta_header_path** (*str*) – path to the "header" file generated by oucass-checklist

merge (*to_add*)
Loads all Profile objects from a pre-existing Profiles into this Profiles. All flights must be from the same location.

Parameters **to_add** (*Profiles*) – the Profiles object to be merged in

read_netCDF (*file_path*)
Re-creates a Profile_Set object which has been saved as a NetCDF

Parameters **file_path** (*string*) – the NetCDF file

Chapter 5

Raw_Profile

Reads data file (JSON or netCDF) and stores the raw data

class Raw_Profile.**Raw_Profile** (*file_path*, *dev=False*, *scoop_id=None*, *nc_level='low'*, *meta_flight_path=None*, *meta_header_path=None*)

Contains data from one file. Data is stored as a pandas DataFrame.

Variables

- **temp** (*tuple*) – temperature as (Temp1, Resi1, Temp2, Resi2, …, time)
- **rh** (*tuple*) – relative humidity as (rh1, T1, rh2, T2, …, time)
- **pos** (*tuple*) – GPS data as (lat, lon, alt_MSL, alt_rel_home, alt_rel_orig, time)
- **pres** (*tuple*) – barometer data as (pres, temp, ground_temp, alt_AGL, time)
- **rotation** (*tuple*) – UAS position data as (VE, VN, VD, roll, pitch, yaw, time)
- **dev** (*bool*) – True if the data is from a developmental flight
- **baro** (*str*) – contains 4-letter code for the type of barom sensor used
- **serial_numbers** (*dict*) – Contains serial number or 0 for each sensor
- **meta** (*Meta*) – processes metadata

get_units ( )

Returns units

is_equal (*other*)

Checks if two Raw_Profiles are the same.

Parameters **other** (*Raw_Profile*) – profile with which to compare this one

pos_data ( )

Gets data needed by the Profile constructor.
rtype: dict return: {"lat":, "lon":, "alt_MSL":, "time":, "units"}

thermo_data ( )

Gets data needed by the Thermo_Profile constructor.
rtype: dict return: {"temp1":, "temp2":, …, "tempj":, "resi1":, "resi2":, …, "resij": , "time_temp": "rh1":, "rh2":, …, "rhk":, "time_rh":, "temp_rh1":, "temp_rh2":, …, "temp_rhk":, "pres":, "temp_pres":, "ground_temp_pres":, "alt_pres":, "time_pres"}

wind_data ( )

Gets data needed by the Wind_Profile constructor.

rtype: list return: {“speed_east”:, “speed_north”:, “speed_down”:, “roll”:, “pitch”:, “yaw”:, “time”:}

Thermo_Profile

Calculates and stores basic thermodynamic parameters

class Thermo_Profile.**Thermo_Profile** (**args*, ***kwargs*)

Contains data from one file.

Variables

- **temp** (*np.array<Quantity>*) – QC'd and averaged temperature
- **mixing_ratio** (*np.array<Quantity>*) – calculated mixing ratio
- **theta** (*np.array<Quantity>*) – calculated potential temperature
- **T_d** (*np.array<Quantity>*) – calculated dewpoint temperature
- **q** (*np.array<Quantity>*) – calculated mixing ratio
- **rh** (*np.array<Quantity>*) – QC'd and averaged relative humidity
- **pres** (*np.array<Quantity>*) – QC'd pressure
- **alt** (*np.array<Quantity>*) – altitude
- **gridded_times** (*np.array<Datetime>*) – times at which processed data exists
- **resolution** (*Quantity*) – vertical resolution in units of time, altitude, or pressure to which the data is calculated

__init__ (**args*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__str__ ( )

Return str(self).

_init2 (*temp_dict*, *resolution*, *file_path=None*, *gridded_times=None*, *gridded_base=None*, *indices=(None, None)*, *ascent=True*, *units=None*, *meta=None*, *nc_level='low'*)

Creates Thermo_Profile object from raw data at the specified resolution.

Parameters

- **temp_dict** (*dict*) – A dictionary of the format {"temp1":, "temp2":, …, "tempj":, "resi1":, "resi2":, …, "resij", "time_temp":, "rh1":, "rh2":, …, "rhk":, "time_rh":, "temp_rh1":, "temp_rh2":, …, "temp_rhk":, "pres":, "temp_pres":, "ground_temp_pres":, "alt_pres":, "time_pres":, "serial_numbers":}, which is returned by Raw_Profile.thermo_data
- **resoltion** (*Quantity*) – vertical resolution in units of altitude or pressure to which the data should be calculated

- **file_path** (*str*) – the path to the original data file WITHOUT the suffix .nc or .json
- **gridded_times** (*np.Array<Datetime>*) – times at which data points should be calculated
- **gridded_base** (*np.Array<Quantity>*) – base values corresponding to gridded_times
- **ascent** (*bool*) – True if data should be processed for the ascending leg of the flight, False if descending
- **units** (*metpy.Units*) – the unit registry created by Profile
- **meta** (*Meta*) – the parent Profile's Meta object
- **nc_level** (*str*) – either 'low', or 'none'. This parameter is used when processing non-NetCDF files to determine which types of NetCDF files will be generated. For individual files for each Raw, Thermo, and Wind Profile, specify 'low'. For no NetCDF files, specify 'none'.

_read_netCDF (*file_path*)

Reads data from a NetCDF file. Called by the constructor.

Parameters **file_path** (*string*) – file name

_save_netCDF (*file_path*)

Save a NetCDF file to facilitate future processing if a .JSON was read.

Parameters **file_path** (*string*) – file name

truncate_to (*new_len*)

Shortens arrays to have no more than new_len data points

Parameters **new_len** – The new, shorter length

Returns None

6.1 Temperature Calibration

Resistance (R) to temperature (T)

```
\begin{aligned} T &= \frac{1}{A + B (log{R}) + C (log{R})^3} \end{aligned}
```

Coefficients are pulled from ./coefs/MasterCoefList.csv on your computer.

Equation from: Greene, B.R. Boundary Layer Profiling Using Rotary-Wing Unmanned Aircraft Systems: Filling the Atmospheric Data Gap. Master's Thesis, The University of Oklahoma, Norman, OK, USA, 2018.

Wind_Profile

Calculates and stores wind parameters

class Wind_Profile.**Wind_Profile** (**args*, ***kwargs*)

Processes and holds wind data from one vertical profile

Variables

- **u** (*list<Quantity>*) – U component of wind
- **v** (*list<Quantity>*) – V-component of wind
- **dir** (*list<Quantity>*) – wind direction
- **speed** (*list<Quantity>*) – wind speed
- **pres** (*list<Quantity>*) – air pressure
- **alt** (*list<Quantity>*) – altitude
- **gridded_times** (*list<Datetime>*) – time of each point
- **resolution** (*Quantity*) – the vertical resolution of the processed data
- **ascent** (*bool*) – is data from the ascending leg of the flight processed? If not, False.

truncate_to (*new_len*)

Shortens arrays to have no more than new_len data points

Parameters **new_len** – The new, shorter length

Returns None

Chapter 8

plotting

fpath_logos

plotting.**contour_height_time** (*profiles, var=['temp'], use_pres=False*)

contourHeightTime creates a filled contour plot of the first element of
var in a time-height coordinate system. If len(var) > 1, it also overlays unfilled contours of the remaining elements. No more than 4 variables can be plotted at once. Accepted variable names are:

- 'theta'
- 'temp'
- 'T_d'
- 'dewp'
- 'r'
- 'mr'
- 'q'
- 'rh'
- 'speed':
- 'u'
- 'v'
- 'dir'
- 'pres'
- 'p'
- 'alt'

Parameters
- **profiles** (*list*) – a list of all profiles to be included in the plot
- **var** (*list<str>*) – names of the variable to be plotted

Return type matplotlib.figure.Figure

Returns the contoured plot

plotting.**plot_skewT** (*profiles, wind_barbs=False, barb_density=10*)

Plots a SkewT diagram. :param list<number> profiles: profiles which contain T_d, press, and temp

data :param bool wind_barbs: if True, plot wind barbs. Requires that profiles contain u, v data. :param int barb_density: n for which every nth barb is plotted :rtype: matplotlib.figure.Figure :return: fig containing a SkewT diagram of the data

utils

profiles.utils.**regrid_base** (**args*, ***kw*)
Used by autodoc_mock_imports.

profiles.utils.**regrid_data** (**args*, ***kw*)
Used by autodoc_mock_imports.

profiles.utils.**temp_calib** ( )
Converts resistance to temperature using the coefficients for the sensor specified OR generalized coefficients if the serial number (sn) is not recognized.

Parameters
- **resistance** (*list<Quantity>*) – resistances recorded by temperature sensors
- **sn** (*int*) – the serial number of the sensor reporting

Return type list<Quantity>

Returns list of temperatures in K

Temperature Calibration

Resistance (R) to temperature (T)

\begin{aligned} T &= \frac{1}{A + B (log{R}) + C (log{R})^3} \end{aligned}

Coefficients are pulled from ./coefs/MasterCoefList.csv on your computer.

Equation from: Greene, B.R. Boundary Layer Profiling Using Rotary-Wing
Unmanned Aircraft Systems: Filling the Atmospheric Data Gap. Master's Thesis, The University of Oklahoma, Norman, OK, USA, 2018.

profiles.utils.**identify_profile** (**args*, ***kw*)
Used by autodoc_mock_imports.

profiles.utils.**qc** (**args*, ***kw*)
Used by autodoc_mock_imports.

profiles.utils.**temp_calib** (**args*, ***kw*)
Used by autodoc_mock_imports.

Chapter 10

Set-Up: Sensor Coefficients

This page attempts to break down the process of specifying your coefficients into manageable steps. If you get stuck, send us a message using the contact form on our home page!

There are 2 ways to host your coefficients so that oucass-profiles can read them. The simplest is to use CSV files in your local file system. Larger teams are likely to prefer the instant updates possible when your coefficients are stored in Azure tables.

10.1 Local File System

10.1.1 Step 1: Start your file structure

First, you're going to want to make a folder somewhere named "coefs". It doesn't matter where you put this folder, just that the name is correct.

```
|coefs
```

10.1.2 Step 2: Assign each platform a unique numerical ID

Inside coefs, create a file named "copterID.csv". This file should contain entries of the format

```
1, name of copter 1
2, name of copter 2
3, name of copter 3
...
```

The numerical ID should be saved to the "SYSID_THISMAV" variable in your JSON file so that each JSON file can be associated with a particular platform.

```
|coefs
 |-copterID.csv
```

10.1.3 Step 3: Assign each sensor to a "scoop"

In profiles, a "scoop" is a collection of sensors. If your platforms have interchangeable sensor loads, this can be really useful. If not, the "scoop" will represent the platform itself.

Label each scoop (or platform) with a single uppercase letter. For each scoop, create a file "scoop<letter>.csv" in coefs. For Scoop A, the file would be called "scoopA.csv". The format should be similar to

validFrom	imet1	imet2	imet3	rh1	rh2	rh3	wind
2019-08-29	57562	57563	58821	1	2	3	944

Any time a sensor is changed, a new line should be added to this file with the date of the change and the new sensor numbers.

```
|coefs
 |-copterID.csv
 |-scoopA.csv
 |-scoopB.csv
 |-...
```

10.1.4 Step 4: Supply the coefficients

Now that profiles will be able to identify which sensors you're using, it's probably a good idea to tell it the coefficients of those sensors. We'll make one more file in the coefs folder, this time named "MasterCoefList.csv". The header for this file should be

SensorType	SerialNumber	ScoopID	Equation	A	B	C	D	Offset	SensorStatus

Each sensor gets its own row. Any field that isn't applicable to a sensor should be filled with "na". Profiles currently supports 3 types of sensors.

Wind

"Wind" is recognized as a sensor type, although the "sensor" in this case is the copter itself. The serial number is the copter's name. If you have interchangeable scoops, the scoop field should be "na". Otherwise, you can set it to the scoop letter associated with the platform.

The equation for wind should be "E1", unless you decide to write your own calibration equation. The default calibration equation requires two coefficients and no offset. The sensor status column is for your personal records.

A row describing a wind sensor should look something like this:

SensorType	SerialNumber	ScoopID	Equation	A	B	C	D	Offset	SensorStatus
Wind	944	na	E1	3.28E+01	-4.50E+00	na	na	na	Active

IMet

The IMet sensor handles temperature. A row describing an IMet sensor could look like this:

SensorType	SerialNumber	ScoopID	Equation	A	B	C	D	Offset	SensorStatus

Imet	45363	na	E2	9.93118592E-04	2.63743049E-04	1.47415476E-07	na	na	Retired

RH

There is not currently a calibration equation for relative humidity - instead, an offset is accepted. A line for an RH sensor should look like this:

SensorType	SerialNumber	ScoopID	Equation	A	B	C	D	Offset	SensorStatus
RH	3	A	na	na	na	na	na	1843	Active

At this point, your coef folder should contain the following files:

```
|coefs
 |-copterID.csv
 |-scoopA.csv
 |-scoopB.csv
 |-...
 |-MasterCoefList.csv
```

10.2 Azure Tables

10.2.1 Step 1: Create an Azure "Storage Account"

Make an account for your team at https://www.portal.azure.com. You can get set up with a free account, but will eventually need to transition to a paid account. Individual team members can make changes to the team account from personal, free accounts.

The easiest way to interact with your Storage Account is through Microsoft Azure Storage Explorer.

10.2.2 Step 2: Assign each platform a unique numerical ID

Each platform (UAS) will need to have an ID, in addition to a name. The ID should be saved to the "SYSID_THISMAV" variable in your JSON file.

Create a new Table in your Storage Account named "Copters". The PartitionKey can be whatever you'd like, or you can leave it set to "default". The CopterID should be assigned to RowKey, with the name of the copter in a new Property (column) called "Name"

10.2.3 Step 3: Assign each sensor to a "scoop"

In oucass-profiles, a "scoop" is a collection of sensors that are used together. If your platforms have interchangable sensor loads, this can be really useful. If not, the "scoop" will represent the platform itself.

Using Azure allows you more freedom in chosing your scoop names. You can use single character names (such as "A") or descriptive names (like "ChemFW1"). Assign these names to the PartitionKey of each row in a new table called "Scoops". The RowKey must be a date in the format YYYYMMDD and should represent the day on which the listed sensors were installed on the scoop. Don't delete outdated entries - oucass-profiles will make sure to use the serial numbers associated with thelatest scoop entry BEFORE

the flight date.

The Parameters of the Scoops table should be “Engineer”, “IMET1”, “IMET2”, …, “RH1”, “RH2”, … The Engineer field can be used to identify the person who created an entry in the case of any discrepancies.

10.2.4 Step 4: Supply the coefficients

The next Table to create in your Storage Account is “MasterCoef”. There should be one row per sensor in your inventory. The PartitionKey identifies the type of sensor (“Imet”, “RH”, or “Wind”), while the RowKey specifies the sensor’s serial number. The serial number “0” is used to specify default coefficients for rough calculations when specific coefficients are not available.

The Parameters “A”, “B”, “C”, “D”, and “Offset” must exist in your table. The lettered columns hold the actual coefficients to convert resistance or voltage to some atmospheric property. “Offset” is used when a sensor (typically an RH sensor) is corrected for bias. Additional Parameters can be added according to your team’s needs.

Chapter 11

Set-up: Configuration

After setting up either your file system or Azure to hold sensor coefficients, you'll need to edit "conf.py". This file will be located in the folder in which oucass-profiles was installed - if you're using Conda, it'll be something like "~/miniconda3/envs/MyEnv/lib/pythonx.y/site-packages/profiles/conf.py". Set the variables in this file so that your coefficients can be found.

1. Position data to wind

```
\begin{aligned} \mathrm{Speed} &= A \sqrt{\tan{[\arccos{(\cos{(\mathrm{PITCH})} -
\cos{(\mathrm{ROLL})})}]}} + B \\ \\ \mathrm{Direction} &=
\arctan{\Bigg[\frac{-\cos{(\mathrm{YAW})}\sin{(\mathrm{ROLL})}}{\sin{(\mathrm{YAW})}\sin{(\mathrm{R
+ \sin{(\mathrm{PITCH})}\cos{(\mathrm{YAW})}\cos{(\mathrm{ROLL})}} \Bigg]}
\end{aligned}
```

1. Thermistor resistance to temperature

```
\begin{aligned} T &= \frac{1}{A + B (log{R}) + C (log{R})^3} \end{aligned}
```

c

m

p

r

t

w

Symbols

A

C

F

G

I

M

P

Q

R

T

W