Norwegian Meteorological Institute

Pyaerocom: Introducing a new calculated variable

Jan Griesfeller

The goal:

- calculate the ratio of concpm10 / concpm25 for all observation networks that provide the data
  - EEA
  - EBAS
  - Marco Polo
  - AirNow
- make it possible via the API

```
DATA_ID = 'EEAAQeRep.v2'
VAR_NAME = ['ratpm10pm25']

def main():
    import pyaerocom.io as pio
    obs_obj = pio.ReadUngridded(DATA_ID)
    obs_data1 = obs_obj.read(vars_to_retrieve=VAR_NAME)
    print(obs_data1)

if __name__ == "__main__":
    main()
```

The concept:

- There's three ways of doing it within pyaerocom:
  1.1. within the obs network specific reading class (that's e.g. how AeronetSun gets from od500aer to od550aer
  1.2. read the variables independently, then co-locate the variable data in time and space and then calculate the wanted variable
  1.3. do it with aeroval (which basically uses 1.2 but the results cannot be used via the API)
- pros / cons:
  - 1.1 works easily if all data fields are ins the same file (not the case for EEA and EBAS; it's the fastest way)
  - using 1.3. the usage would be limited to the aeroval web page. I could not find an example for calculated obs vars although the infrastructure seems to be there
  - 1.2 is the most universal and works for all obs networks. It's the slowest and uses most RAM

What I found:

- The functions to calculate the results are defined in the reading class:
- [read_eea_aqerep_base.py]

```
AUX_REQUIRES = {
    "vmro3": ["conco3"],
    "vmrno2": ["concno2"],
    RATPM10PM25_NAME: ["concpm10", "concpm25"],
}

AUX_FUNS = {
    "vmro3": NotImplementedError(),
    "vmrno2": NotImplementedError(),
    RATPM10PM25_NAME: compute_ratpm10pm25,
}
```

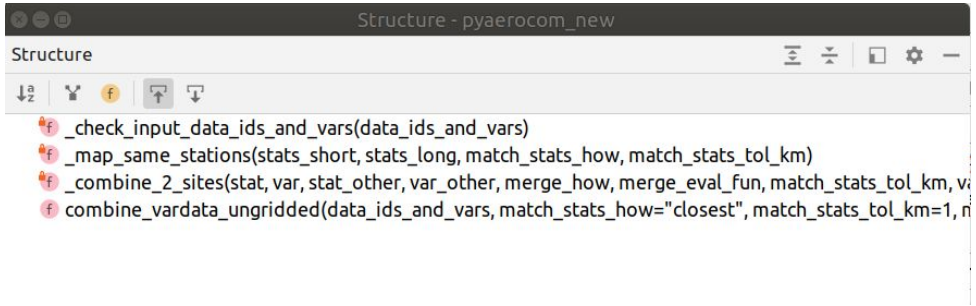
The calculation method

- The existing calculation routines are usually in **pyaerocom/pyaerocom/aux_var_helpers.py**
- The code as it is imposes using a pandas DataFrame as data structure
  - NO UNIT handling!
- The API is not clearly defined as there's methods using dict like objects, ndarrays / floats or the StationData object as input data

```
if isinstance(data, pandas.core.frame.DataFrame):
    # this is used if the variable calculation is done via the API
    data[outvar_name] = data[concpm10_name] / data[concpm25_name]
    return data
else:
    raise NotImplementedError(
        f"{__name__}: Can only handle inputdata of type pandas.core.frame.DataFrame"
    )
```

Logic to glue everything together

- The components to do what I wanted seemed to be existing in pyaerocom (selecting equal stations, equal time steps, do the actual calculations, etc) but the logic to get it called seems odd. [pyaerocom/combine_vardata_ungridded.py]



- I could not find a usage of the existing logic (old aerocom-evaluation?), but I did not want to break potential usages.
- Recreated the existing data structures to use the existing logic

The data structure

```
try:
    aux_info = self.post_compute[data_id]
except KeyError:
    self.post_compute[data_id] = {}
    self.post_compute[data_id]["data_id"] = data_id
    #
    self.post_compute[data_id]["aux_requires"] = {}
    self.post_compute[data_id]["aux_merge_how"] = {}
    self.post_compute[data_id]["aux_units"] = {}
    self.post_compute[data_id]["aux_funs"] = {}
    # # to make sure the API reading logic is called later on
    # (and not e.g. aeroval)
    self.post_compute[data_id]["aux_flag"] = True
    # The getattr calls fail without the following line
    reader = self.get_lowlevel_reader(data_id)
    for var in vars_to_retrieve:
        self.post_compute[data_id]["aux_requires"][var] = {}
        self.post_compute[data_id]["aux_requires"][var][data_id] = {}
        self.post_compute[data_id]["aux_requires"][var][data_id] = getattr(
            self._readers[data_id], "AUX_REQUIRES"
        )
        # Supported are 'combine', 'mean' and 'eval'
        self.post_compute[data_id]["aux_merge_how"][var] = "eval"
        self.post_compute[data_id]["aux_units"][var] = "1"

        self.post_compute[data_id]["aux_funs"][var] = {}
        self.post_compute[data_id]["aux_funs"][var] = getattr(
            self._readers[data_id], "AUX_FUNS"
        )[var]
        aux_info = self.post_compute[data_id]
```

Conclusions

- Hardly any developer documentation exists!
- Things are too complicated and need too many resources! (e.g. ungridded reading):
  - data is read into point cloud
  - out of that pandas time series are created and mostly used afterwards (inside the UngriddedData object)
  - for the calculated variables all operations are done using a pandas DataFrame
  - then the point cloud is created again
- absolutely no parallelism (despite what e.g. numpy does internally already)
- get rid of human thinking! (e.g. df.dropna() makes sense only at the very end because in contrary to human thinking it increases max RAM usage (for df recreation) and does not save RAM)

Questions?