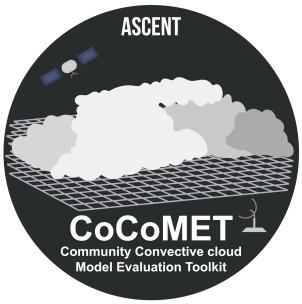
Community Convective cloud Model Evaluation Toolkit (CoCoMET) Developer Guide

Travis Hahn[1], Dié Wang[2], Hershel Weiner[3], Calvin Brooks[4], Jie Xi Li[5], and Siddhant Gupta[6]

[1]Department of Statistics, The Pennsylvania State University
[2]Environmental and Climate Sciences Department, Brookhaven National Laboratory
[3]Physics and Astronomy Department, University of Hawaii
[4]Physics, Applied Physics, and Astronomy Department, Rensselaer Polytechnic Institute
[5]Applied Mathematics & Statistics, Stony Brook University
[6]Environmental Sciences Division, Argonne National Laboratory

January 2025



A toolkit of the Advanced Study of Cloud and Environment iNTerations (ASCENT) program.

Contents

Acronyms

ARM	Atmospheric Raditaion Measurement
NCAR	National Center for Atmospheric Research
CoCoMET	Community Convective cloud Model Evaluation Toolkit
CoCoMET-US	Community Convective cloud Model Evaluation Toolkit Unified Specification
WA	Vertical Velocity at Mass Points
TB	Brightness Temperatire
tobac	Tracking and Object-Based Analysis of Clouds
IDE	Integrated Developer Environment
DC	Data Cell
CC	Convective Cell
BNL	Brookhaven National Laboratory

Document Changes to Come

- Update CoMET Logo
- Fix CoMET-US formatting and references to CoMET-UDAF
- Add Datatypes to CoMET-US
- Add information about adding analysis functions
- Add info for adding a tracker
- Add info for adding a new input type

1 Introduction to CoMET

Hello, National Center for Atmospheric Research (NCAR), Atmospheric Raditaion Measurement (ARM), yep ARM Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. and Brookhaven National Laboratory (BNL)

2 The CoMET-US

TODO: Update words here

Unified Data Analysis Format UDAF dictates/defines how output data from various trackers should be formatted. This facilitates the analysis module of CoMET to prevent mismatched information being used in the analysis module of CoMET. The first thing which is uniformly defined is the structure of the input configuration file which hereafter will be referred to as the "CONFIG".

2.1 The CONFIG

The CONFIG should either be stored as a .yaml file or created as a python dictionary object– please see the boilerplate.yml file to see a proper setup. The CONFIG is broken down into two primary sections with a number of subsections:

1. Parameters relating to the running of CoMET. These include the following:
   (a) verbose [bool]: Whether or not to output text during the running of CoMET.
   (b) parallel_processing [bool]: Whether or not to use multiple cores for processes when available. This would primarily be invoked when gridding radar data and processing multiple input data types simultaneously.
   (c) max_cores [int]: How many cores CoMET may use if parallel_processing is True.
2. The observation type which will be another dictionary with the following values:
   (a) path_to_data [string]: A glob-like path (e.g., "/usr/data/WRF/wrfout_d02*") to the raw data files for your observation type
   (b) feature_tracking_var [string]: A string designating which variable to use for feature identification and linking
   (c) segmentation_var [string]: A string designating which variable to use for segmentation
   (d) gridding [optional, dict]: The parameters set for the gridding of radar data. This should be a dictionary with parameter values set for the `Py-ART` gridding function.
   (e) bounds [optional, array]: Sets the bounds of an observation type. Only works for observations including NEXRAD and GOES. It should be defined in the following order: `[min_longitude, max_longitude, min_latitude, max_latitude]`

3. The tracker to be used for this observation type. This should be another dictionary with subsections determined by the selected tracker.
4. The analysis section: another dictionary containing the variable to be calculated as the key and a dictionary of the parameters necessary as the value.

Here is an example CONFIG for a simple Community Convective cloud Model Evaluation Toolkit (CoCoMET) run:

```
verbose: True # Whether to use verbose output
parallel_processing: True # [bool] Whether or not to use parallel processing for certain tasks
max_cores: 32 # Number of cores to use if parallel_processing==True; Enter None for unlimited

wrf:
   path_to_data: "/D3/data/thahn/wrf/wrfout_2023_07_09/wrfout_d02*"

   is_idealized: False
   min_frame_index: 10 # 0-based indexing, inclusive
   max_frame_index: 70 # 0-based indexing, inclusive

   feature_tracking_var: "DBZ"
   segmentation_var: "DBZ"

   tobac:
      feature_id:
         threshold: [30,40,50,60]
         target: "maximum"
         position_threshold: "weighted_diff"
         sigma_threshold: 0.5
         n_min_threshold: 4

      linking:
         method_linking: "predict"
         adaptive_stop: 0.2
         adaptive_step: 0.95
         order: 1
         subnetwork_size: 10
         memory: 1
         v_max: 20

      segmentation_2d:
         height: 2 # km
         method: "watershed"
         threshold: 15

      segmentation_3d:
         method: "watershed"
         threshold: 15

      analysis: # Optional
```

```
merge_split: { variable: "DBZ" }
var_max_height: { variable: "DBZ", cell_footprint_height: 2, threshold: 15 }
area-low: { height: 2 }
area-high: { height: 4, threshold: 15 }
volume: { threshold: 15 }
volume-high: { threshold: 30 }
```

2.2 Tracking Output Structure

It is necessary to define a uniform output structure which all trackers should be converted to for the purpose of later analysis. This section defines the primary components of tracker output including feature identification, linking, and segmentation output.

2.2.1 Feature Identification Output

Feature identification output from trackers should be converted to a `GeoPandas geodataframe` where each row should contain information regarding exactly one detected feature. The row should contain the following information in this order:

1. "frame" [int]: The index of the time step the feature was identified at. Should be an integer ranging from 0 to N where N is the number of time steps in the input data source.
2. "time" [`pandas Timestamp`]: A `pandas Timestamp` object indicating the time of feature detection based off of the input time field.
3. "feature_id" [int]: A unique id value assigned to each detected feature. Should be an integer ranging from 0 to N where N is the total number of features detected.
4. "south_north" [float]: The y index of the detected feature determined by the input field. May not be an integer for some trackers such as tobac.
5. "west_east" [float]: The x index of the detected feature determined by the input field. May not be an integer for some trackers such as tobac.
6. "up_down" [optional, float]: The z index of the detected feature determined by the input field. May not be an integer for some trackers such as tobac. May not be present if only doing 2D tracking.
7. "latitude" [float]: The exact latitude value of the location of the detected feature.
8. "longitude" [float]: The exact longitude value of the location of the detected feature.
9. "projection_x" [float]: The projection x value of the feature. Should be in meters.
10. "projection_y" [float]: The projection y value of the feature. Should be in meters.
11. "altitude" [optional, float]: The altitude of the identified feature in meters. May not be present if only doing 2D tracking.
12. "geometry" [`Geopandas point`]: The `point` location of the features determined using lat/long values generated by `GeoPandas`.

2.2.2 Linking Output

Tracking or "linking" output from the tracker should also be a `GeoPandas geodataframe` where each row should contain information regarding exactly one identified feature. The row should

contain the following information (can be seen as an extension of the feature identification output):

1. "frame" [int]: The index of the time step the feature was identified at. Should be an integer ranging from 0 to N where N is the number of time steps in the input data source.
2. "time" [`pandas Timestamp`]: A `pandas Timestamp` object indicating the time of feature detection based off of the input time field.
3. "lifetime" [`pandas Timedelta`]: A `pandas Timedelta` object indicating the time since the cell was first tracked.
4. "lifetime_percent" [float]: A float indicating the percentage of the cell's lifetime the current row is. I.e. If the cell lasts 30 minutes, the 15 minute row would be 0.5. If Convective Cell (CC) life is only one frame, will equal -1.
5. "feature_id" [int]: A unique id value assigned to each detected feature. Should be an integer ranging from 0 to N where N is the total number of features detected.
6. "cell_id" [int]: A unique id value assigned to each detected cell track. Should be an integer ranging from 0 to N where N is the total number of CCs tracked.
7. "south_north" [float]: The y index of the detected feature determined by the input field. May not be an integer for some trackers such as tobac.
8. "west_east" [float]: The x index of the detected feature determined by the input field. May not be an integer for some trackers such as tobac.
9. "up_down" [optional, float]: The z index of the detected feature determined by the input field. May not be an integer for some trackers such as tobac. May not be present if only doing 2D tracking.
10. "latitude" [float]: The exact latitude value of the location of the detected feature.
11. "longitude" [float]: The exact longitude value of the location of the detected feature.
12. "projection_x" [float]: The projection x value of the feature. Should be in meters.
13. "projection_y" [float]: The projection y value of the feature. Should be in meters.
14. "altitude" [optional, float]: The altitude of the identified feature in meters. May not be present if only doing 2D tracking.
15. "geometry" [`Geopandas point`]: The `point` location of the features determined using lat/long values generated by `GeoPandas`.

2.2.3 Segmentation Output

5.2.3. The segmentation step of each tracker should be an `xarray Dataset` containing two data variables with the following dimensions in this order:

1. "time" [`numpy datetime64`]: A list of `numpy datetime64` objects containing all of the time steps of the input data.
2. "up_down" [int]: The z index of each Data Cell (DC) in the `Dataset`, should be an integer for each value. May not be present for 2D segmentation.

3. "south_north" [int]: The y index of each DC in the `Dataset`, should be an integer for each value.
4. "west_east" [int]: The x index of each DC in the `Dataset`, should be an integer for each value.

And the following coordinates in no particular order:

1. "projection_x_coordinate" [float]: The projection x location of each DC in the `Dataset` given in meters.
2. "projection_y_coordinate" [float]: The projection y location of each DC in the `Dataset` given in meters.
3. "altitude" [float]: The altitude in meters of each DC in the `Dataset`.
4. "latitude" [float]: The latitude of each DC in the `Dataset` (Should be 2 dimensional `[south_north, west_east]`).
5. "longitude" [float]: The longitude of each DC in the `Dataset` (Should be 2 dimensional `[south_north, west_east]`).

The first data variable will be `Feature_Segmentation` where the field is labeled with feature ids according to the segmentation. The second variable will be a `Cell_Segmentation` which will be the same as the `Feature_Segmentation` but instead of each point in the output field being labeled with the feature ids, they will be replaced with the respective CC id of that feature. Locations on the grid without a CC should be marked as -1 to coincide with the standard established in the sections above.

2.3 Analysis Products

Community Convective cloud Model Evaluation Toolkit Unified Specification (CoCoMET-US) analysis products used in variable calculations.

2.3.1 The Analysis Object

The analysis object is a dictionary consisting of the following items:

1. "tracking_xarray" [`xarray Dataset/DataArray`]: An `xarray Dataset/DataArray` which contains the variables used to perform the tracking and potentially other variables. Will be different depending on the type of data used.
2. "segmentation_xarray" [`xarray Dataset/DataArray`]: An `xarray Dataset/DataArray` which contains the variables used to perform the segmentation on. Will be different depending on the type of data used.
3. "UDAF_features" [`Geopandas geodataframe`]: A UDAF_features compliant `Geopandas geodataframe`.
4. "UDAF_linking" [`Geopandas geodataframe`]: A UDAF_linking compliant `Geopandas geodataframe`.
5. "UDAF_segmentation_2d" [`xarray Dataset`]: A UDAF_segmentation_2d compliant `xarray Dataset`.
6. "UDAF_segmentation_3d" [`xarray Dataset`]: A UDAF_segmentation_3d compliant `xarray Dataset`.

2.4 Outputs to User

TODO: Define the return structure of CoMET_start and all elements therein

3 General Guidelines

When making any changes to **CoMET!** (**CoMET!**), whether that be adding a new analysis function or adding a whole new tracker, there are some general guidelines we should follow. They are laid out here, in the [semi-]order (sometimes it may not be necessary to rigidly follow this routine, but it is recommended you do) they should be completed:

- When importing python packages into the file you are working on, try your best to collect the imports all at the top of the document. Refrain from putting imports inside of function definitions unless absolutely necessary.
- Whenever you define a new python function, it should have type hints indicating the potential input types and potential output types. For example:

```
def extract_arm_product(
analysis_object: dict, path_to_files: str, variable_names: list[str],
**args: dict) -> xr.Dataset:
```

- Whenever a new python function is defined, it should have a docstring which indicates more information about the input arguments and the return values. These docstrings should follow the `Sphinx` formatting scheme which the Spyder Integrated Developer Environment (IDE) does automatically. For the above function definition, the docstring looks like this:

```
"""
Parameters
----------
analysis_object : dict
   A CoMET-UDAF standard analysis object containing at least UDAF_tracks.
path_to_files : str
   A glob-like path to the ARM product output.
variable_name : list[str]
   Case sensitive name of variable you want to extract from the ARM data

Returns
-------
output_data : xarray.core.dataset.Dataset
   An xarray Dataset with the following: frame, tracking_time, arm_time,
   time_delta, closest_feature_id (km), variable_names list
"""
```

- Whenever a new python function is defined, it should fall into some kind of test (we use the `pytest` testing suite). There will be more information on this in future sections, however, functions which primarily deal with the loading/tracking step will mostly fall under the **functional tests** whereas those which deal with the analysis or post-processing will fall under the **unit tests**.
- Now that your code is prepared, there a few processes we should follow. The first of which is to reformat your code so it is in line with the PEP8 formatting standards. We have a few packages which will automatically do this. The first one is going to be `isort`. This package will reorder and reshape your import statements so they are more readable. To use it, simply do `python -m isort dir --profile black` from the command line where

`dir` is the directory which contains the `__init__.py` file. For **CoMET!**, this will just be the `CoMET/` directory.

- Now to reformat the body of the code, we are going to be using the `black` package. This is also simple, as you just need to do `python -m black dir` from the command line. **WARNING:** Do **not** interrupt `black` during its runtime as this can result in a complete deletion of your files–luckily, `black` usually takes < 1 second to run.
- Now that our code is nicely formatted, we need to add the automatically generated documentation (this is why the docstrings were important). Firstly, if you created a new file in your changes, you need to edit the `__all__` variable in **CoMET!**'s `__init__.py` file. All you need to do is add the name of the new file in quotes to the array. So like this:

```
__all__ = [
"user_interface_layer",
.
.
.
"user_utils",
"your_file_name"
]
```

- Now we can generate the documentation. All we need to do is run the following command from the command line:

  `python -m pdoc dir --logo "https://url.to.ascent.logo.here" -o ./docs`

- Once all of this is done, and probably intermittently as well, we need to run our tests. As long as you kept the tests within the `dir/tests/` directory, you should be able to automatically run all tests with the following command (make sure they pass before committing!):

  `python -m pytest`

- Now we can build our package for distribution. Ensuring we have the `build` package installed, all we have to do is run `python -m build dir` and this will handle everything. You are now ready to commit your updates to **CoMET!**! Make sure you do not commit to the `master` branch first, you should always commit to `testing` branch or your own branch.

4 Adding an Analysis Function

Suppose we want to add another function to the analysis module of **CoMET!**, this would be the case when we wish to calculate some properties or quantities of interest relating to the tracking results. Importantly, we restrict the bounds of what the function we want to add can do based off of where it fits into the **CoMET!** paradigm. Namely, we have two options. The first is an **internal function**, this is something which can be evaluated during the normal **CoMET!** operating procedure (i.e. The **CoMET!** start function), i.e. within the analysis step. Secondly, we have a **post-processing function** which is something which operates outside of the **CoMET!** run step and acts mainly as a function to help users. An example of the first type is say we want to calculate the area of a cell, this is something that can be done within the primary analysis module. Conversely, an example of the second function type is something in which we don't have enough information to do during the analysis step. Say we want to 'link' together Tracking

and Object-Based Analysis of Clouds (tobac) Vertical Velocity at Mass Points (WA) and Brightness Temperatire (TB) tracking results to see how their centroids move. Since we cannot track with both variables simultaneously (as this would be unnecessarily difficult), we would take in **CoMET!** output from two different runs to apply this function.

In the following subsections, we describe how you should go about adding each of the function types in a step by step manner.

4.1 Adding an Internal Function

TODO

4.2 Adding a Post-Processing Function

TODO

5 Adding a Tracker

TODO

6 Adding a New Input Type

TODO

This should be it.