MARBLE Documentation

Release 0.1.0

Jeremy McGibbon

Jul 03, 2019

Contents:

Machine Assisted Boundary Layer Emulation is a neural network based parameterization for weather and climate models, using the Sympl framework. It requires Python 3.

CHAPTER 1

Installation

1.1 Stable release

To install MARBLE, run this command in your terminal:

```
$ pip install marble
```

This is the preferred method to install MARBLE, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2 From sources

The sources for MARBLE can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/mcgibbon/marble
```

Or download the tarball:

```
$ curl  -OL https://github.com/mcgibbon/marble/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 2

Quickstart

For installation instructions see *Installation*.

Once installed, check out some of the scripts in the examples folder. These should run from any directory you put them in.

A good starting place is the single column model in scm.py.

If you haven't used Sympl before, you may want to read the Sympl documentation, or at least the Sympl quickstart.

CHAPTER 3

Usage

To use MARBLE in a project:

```
import marble
```

MARBLE uses the Sympl framework. You can read more in the Sympl documentation. MARBLE comes with code examples, which can be accessed from your local installation or the MARBLE github repo.

3.1 Aliases

As of writing this documentation, one shortcoming of Sympl is the need to explicitly write the long name of any quantities that are accessed from a state dictionary in the run script or analysis code. This package adds some helper tools that avoid this requirement, by allowing you to refer to long names using *aliases* that you register at the top of your run script. This retains the benefit of having quantities explicitly defined, because anyone reading your code can look at the top where you register aliases to figure out what your short aliases mean.

For example:

```
import marble
import sympl

marble.register_alias('rt', 'total_water_mixing_ratio')
# or
# marble.register_alias_dict({'rt': 'total_water_mixing_ratio'})

state = {
    'total_water_mixing_ratio': sympl.DataArray(0., dims=[], attrs={'units': 'kg/kg'})
}

state = marble.AliasDict(state)
print(state['rt'])  # gets state['total_water_mixing_ratio']
```

marble.**register_alias**(*alias*, *long_name*)

marble.**register_alias_dict**(*alias_dict*)

class marble.**AliasDict**(**args*, ***kwargs*)

3.2 Initialization

State initialization is not performed by the MARBLE module, but we do give an example initialization code. To call the modules, you need to create a state that has all the required quantities with defined dimensions and units.

One thing to keep in mind, which we will discuss more below, is that MARBLE runs using principal components of its vertically-resolved quantities. Those principal components are pre-defined, and assume that their height-resolved inputs are on a 3km, 20-point equidistant grid with points at 0km and 3km.

3.3 Decomposition

As we just said, MARBLE runs using principal components of its vertically-resolved quantities. Those principal components are pre-defined, and assume that their height-resolved inputs are on a 3km, 20-point equidistant grid with points at 0km and 3km.

When MARBLE is run, it operates on principal components of vertically-resolved quantities. This means that before integration, the state needs to be converted into principal components, and after integration they need to be converted back to height coordinates before plotting or analysis.

To convert between height and principal components, we provide two helper functions that operate on one quantity at a time, and three Sympl components which operate on commonly-grouped quantities.

marble.**convert_height_to_principal_components**(*array*, *basis_name*, *subtract_mean=True*)
Converts a numpy array from height coordinates on a 20-point equidistant grid from 0 to 3km (inclusive) into principal components required by MARBLE.

Parameters

- **array** – numpy array whose final dimension is of size 20
- **basis_name** – short alias name of the quantity whose principal components to use. For example, 'rt', 'sl', 'cld', 'rcld', 'rrain', or 'w'.
- **subtract_mean** – whether to subtract the mean vertical profile of the basis quantity from the numpy array before converting into principal components. Generally this is True if you are converting the basis quantity itself, and False if you are converting a difference to apply to the basis quantity (such as a tendency).

Returns

numpy array whose final dimension length is equal to the number of principal components used for hte basis quantity.

Return type return_array

marble.**convert_principal_components_to_height**(*array*, *basis_name*, *add_mean=True*)
Converts a numpy array from principal components as used by MARBLE to height coordinates on a 20-point equidistant grid from 0 to 3km (inclusive).

Parameters

- **array** – numpy array whose final dimension is principal component number
- **basis_name** – short alias name of the quantity whose principal components are used. For example, 'rt', 'sl', 'cld', 'rcld', 'rrain', or 'w'.

- **add_mean** – whether to add in the mean vertical profile of the basis quantity from the numpy array after converting to height coordinates. Generally this is True if you are converting the basis quantity itself, and False if you are converting a difference applied to the basis quantity (such as a tendency).

Returns numpy array whose final dimension length is 20.

Return type return_array

class marble.**InputHeightToPrincipalComponents**

Converts MARBLE's vertically-resolved inputs from height coordinates to principal components.

Input Properties:

liquid_water_static_energy: alias: sl, dims: ['*', 'z_star'], units: J/kg,

total_water_mixing_ratio: alias: rt, dims: ['*', 'z_star'], units: kg/kg,

vertical_wind: alias: w, dims: ['*', 'z_star'], units: m/s,

Diagnostic Properties:

liquid_water_static_energy_components: alias: sl_latent, dims: ['*', 'sl_latent'], units: ,

total_water_mixing_ratio_components: alias: rt_latent, dims: ['*', 'rt_latent'], units: ,

vertical_wind_components: alias: w_latent, dims: ['*', 'w_latent'], units: ,

class marble.**InputPrincipalComponentsToHeight**

Converts MARBLE's vertically-resolved inputs from principal components to height coordinates.

Input Properties:

liquid_water_static_energy_components: alias: sl, dims: ['*', 'sl_latent'], units: ,

total_water_mixing_ratio_components: alias: rt, dims: ['*', 'rt_latent'], units: ,

vertical_wind_components: alias: w, dims: ['*', 'w_latent'], units: ,

Diagnostic Properties:

liquid_water_static_energy: alias: sl, dims: ['*', 'z_star'], units: J/kg,

total_water_mixing_ratio: alias: rt, dims: ['*', 'z_star'], units: kg/kg,

vertical_wind: alias: w, dims: ['*', 'z_star'], units: m/s,

class marble.**DiagnosticPrincipalComponentsToHeight**

Converts MARBLE's vertically-resolved diagnostic outputs from principal components to height coordinates.

Input Properties:

cloud_water_mixing_ratio_components: alias: rcld, dims: ['*', 'rcld_latent'], units: ,

rain_water_mixing_ratio_components: alias: rrain, dims: ['*', 'rrain_latent'], units: ,

cloud_fraction_components: alias: cld, dims: ['*', 'cld_latent'], units: ,

clear_sky_radiative_heating_rate_components: alias: sl_rad_clr, dims: ['*', 'sl_latent'], units: hr^-1,

Diagnostic Properties:

cloud_water_mixing_ratio: alias: rcld, dims: ['*', 'z_star'], units: ,

rain_water_mixing_ratio: alias: rrain, dims: ['*', 'z_star'], units: ,

cloud_fraction: alias: cld, dims: [‘*’, ‘z_star’], units: ,

clear_sky_radiative_heating_rate: alias: sl_rad_clr, dims: [‘*’, ‘z_star’], units: degK hr^-1,

3.4 Forcing

We use an extremely simple component to apply horizontal advective forcings that are defined in the state as tendencies to the prognostic quantities. The horizontal advective forcings need to be defined in principal component space. This can be achieved using *marble.convert_height_to_principal_components()*.

class marble.**LatentHorizontalAdvectiveForcing**(*TendencyComponent*)

MARBLE component which applies advective forcings in latent space (inputs and outputs denormalized principal components) without converting to or from the real height coordinate.

Works by applying an advective tendency already loaded and specified in the model state.

3.5 MARBLE

MARBLE itself is contained in a *TendencyComponent*. Note that the surface latent and sensible heat fluxes should be expressed as downward values, as in the flux into the surface.

class marble.**LatentMarble**(*tendencies_in_diagnostics=False*, *name=None*)

MARBLE component which works in latent space (inputs and outputs denormalized principal components) without converting to or from the real height coordinate.

Input Properties:

liquid_water_static_energy_components: alias: sl, dims: [‘*’, ‘sl_latent’], units: ,

total_water_mixing_ratio_components: alias: rt, dims: [‘*’, ‘rt_latent’], units: ,

vertical_wind_components: alias: w, dims: [‘*’, ‘w_latent’], units: ,

liquid_water_static_energy_at_3km: alias: sl_domain_top, dims: [‘*’], units: J/kg,

total_water_mixing_ratio_at_3km: alias: rt_domain_top, dims: [‘*’], units: kg/kg,

surface_latent_heat_flux: alias: lhf, dims: [‘*’], units: W/m^2,

surface_sensible_heat_flux: alias: shf, dims: [‘*’], units: W/m^2,

surface_temperature: alias: sst, dims: [‘*’], units: degK,

mid_cloud_fraction: alias: cldmid, dims: [‘*’], units: ,

high_cloud_fraction: alias: cldhigh, dims: [‘*’], units: ,

downwelling_shortwave_radiation_at_top_of_atmosphere: alias: swdn_toa, dims: [‘*’], units: W/m^2,

downwelling_shortwave_radiation_at_3km: alias: swdn_tod, dims: [‘*’], units: W/m^2,

surface_air_pressure: alias: p_surface, dims: [‘*’], units: Pa,

rain_water_mixing_ratio_at_3km: alias: rrain_domain_top, dims: [‘*’], units: kg/kg,

Diagnostic Properties:

cloud_water_mixing_ratio_components: alias: rcld, dims: [‘*’, ‘rcld_latent’], units: ,

rain_water_mixing_ratio_components: alias: rrain, dims: [‘*’, ‘rrain_latent’], units: ,

cloud_fraction_components: alias: cld, dims: [‘*’, ‘cld_latent’], units: ,

clear_sky_radiative_heating_rate_components: alias: sl_rad_clr, dims: [‘*’, ‘sl_latent’], units: hr^-1,

low_cloud_fraction: alias: cldlow, dims: [‘*’], units: ,

surface_precipitation_rate: alias: precip, dims: [‘*’], units: mm/hr,

column_cloud_water: alias: ccw, dims: [‘*’], units: kg/m^2,

height: alias: z, dims: [‘z_star’], units: m,

Tendency Properties:

liquid_water_static_energy_components: alias: sl, dims: [‘*’, ‘sl_latent’], units: hr^-1,

total_water_mixing_ratio_components: alias: rt, dims: [‘*’, ‘rt_latent’], units: hr^-1,

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/mcgibbon/marble/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

MARBLE could always use more documentation, whether as part of the official MARBLE docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mcgibbon/marble/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *marble* for local development.

1. Fork the *marble* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/marble.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv marble
$ cd marble/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 marble tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated.
3. The pull request should work for Python 3.4, 3.5, and 3.6. Check https://travis-ci.org/mcgibbon/marble/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 5

Credits

5.1 Development Lead

- Jeremy McGibbon <mcgibbon@uw.edu>

5.2 Contributors

None yet. Why not be the first?

5.3 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 6

History

6.1 0.1.0 (2019-05-30)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index