mosartwmpy

Release latest

Travis B. Thurber

Dec 29, 2022

CONTENTS

Date: Dec 29, 2022 **Version**: 0.5.1

Useful links: Source Repository | Issues & Ideas

mosartwmpy is a Python translation of MOSART-WM, a water routing and reservoir management model written in Fortran.

Getting started

Get to know the *mosartwmpy* model.

Getting started

Tutorial

Follow along with this Jupyter notebook to learn the ropes of *mosartwmpy*.

Tutorial

Tips & tricks

Learn about ways to manage Python virtual environments.

Virtual environments

API reference

A detailed description of the *mosartwmpy* API.

API

CHAPTER

ONE

MOSARTWMPY

`mosartwmpy` is a python translation of MOSART-WM, a model for water routing and reservoir management written in Fortran. The original code can be found at IWMM and E3SM, in which MOSART is the river routing component of a larger suite of earth-science models. The motivation for rewriting is largely for developer convenience – running, debugging, and adding new capabilities were becoming increasingly difficult due to the complexity of the codebase and lack of familiarity with Fortran. This version aims to be intuitive, lightweight, and well documented, while still being highly interoperable. For a quick start, check out the Jupyter notebook tutorial!

CHAPTER

TWO

GETTING STARTED

Ensure you have Python >= 3.7 available (consider using a virtual environment, see the docs here for a brief tutorial), then install `mosartwmpy` with:

```
pip install mosartwmpy
```

Alternatively, install via conda with:

```
conda install -c conda-forge mosartwmpy
```

Download a sample input dataset spanning May 1981 by running the following and selecting option 1 for "tutorial". This will download and unpack the inputs to your current directory. Optionally specify a path to download and extract to instead of the current directory.

```
python -m mosartwmpy.download
```

Settings are defined by the merger of the `mosartwmpy/config_defaults.yaml` and a user specified file which can override any of the default settings. Create a `config.yaml` file that defines your simulation (if you chose an alternate download directory in the step above, you will need to update the paths to point at your data):

config.yaml

```
simulation:
  name: tutorial
  start_date: 1981-05-24
  end_date: 1981-05-26

grid:
  path: ./input/domains/mosart_conus_nldas_grid.nc

runoff:
  read_from_file: true
  path: ./input/runoff/runoff_1981_05.nc

water_management:
  enabled: true
  demand:
    read_from_file: true
    path: ./input/demand/demand_1981_05.nc
  reservoirs:
    enable_istarf: true
    parameters:
      path: ./input/reservoirs/reservoirs.nc
```

(continues on next page)

(continued from previous page)

```
    dependencies:
      path: ./input/reservoirs/dependency_database.parquet
    streamflow:
      path: ./input/reservoirs/mean_monthly_reservoir_flow.parquet
    demand:
      path: ./input/reservoirs/mean_monthly_reservoir_demand.parquet
```

mosartwmpy implements the Basic Model Interface defined by the CSDMS, so driving it should be familiar to those accustomed to the BMI. To launch the simulation, open a python shell and run the following:

```python
from mosartwmpy import Model

# path to the configuration yaml file
config_file = 'config.yaml'

# initialize the model
mosart_wm = Model()
mosart_wm.initialize(config_file)

# advance the model one timestep
mosart_wm.update()

# advance until the `simulation.end_date` specified in config.yaml
mosart_wm.update_until(mosart_wm.get_end_time())
```

CHAPTER

THREE

MODEL INPUT

Input for `mosartwmpy` consists of many files defining the characteristics of the discrete grid, the river network, surface and subsurface runoff, water demand, and dams/reservoirs. Currently, the gridded data is expected to be provided at the same spatial resolution. Runoff input can be provided at any time resolution; each timestep will select the runoff at the closest time in the past. Currently, demand input is read monthly but will also pad to the closest time in the past. Efforts are under way for more robust demand handling.

Dams/reservoirs require four different input files: the physical characteristics, the average monthly flow expected during the simulation period, the average monthly demand expected during the simulation period, and a database mapping each GRanD ID to grid cell IDs allowed to extract water from it. These dam/reservoir input files can be generated from raw GRanD data, raw elevation data, and raw ISTARF data using the provided utility. The best way to understand the expected format of the input files is to examine the sample inputs provided by the download utility: `python -m mosartwmpy.download`.

3.1 multi-file input

To use multi-file demand or runoff input, use year/month/day placeholders in the file path options like so:

- If your files look like `runoff-1999.nc`, use `runoff-{Y}.nc` as the path
- If your files look like `runoff-1999-02.nc`, use `runoff-{Y}-{M}.nc` as the path
- If your files look like `runoff-1999-02-03`, use `runoff-{Y}-{M}-{D}.nc` as the path, but be sure to provide files for leap days as well!

CHAPTER

FOUR

MODEL OUTPUT

By default, key model variables are output on a monthly basis at a daily averaged resolution to `./output/<simulation name>/<simulation name>_<year>_<month>.nc`. See the configuration file for examples of how to modify the outputs, and the `./mosartwmpy/state/state.py` file for state variable names.

Alternatively, certain model outputs deemed most important can be accessed using the BMI interface methods. For example:

```python
from mosartwmpy import Model

mosart_wm = Model()
mosart_wm.initialize()

# get a list of model output variables
mosart_wm.get_output_var_names()

# get the flattened numpy.ndarray of values for an output variable
supply = mosart_wm.get_value_ptr('supply_water_amount')
```

CHAPTER

FIVE

SUBDOMAINS

To simulate only a subset of basins (defined here as a collection of grid cells that share the same outlet cell), use the configuration option `grid -> subdomain` (see example below) and provide a list of latitude/longitude coordinate pairs representing each basin of interest (any single coordinate pair within the basin). For example, to simulate only the Columbia River basin and the Lake Washington regions, one could enter the coordinates for Portland and Seattle:

config.yaml

```
grid:
  subdomain:
    - 47.6062,-122.3321
    - 45.5152,-122.6784
  unmask_output: true
```

By default, the output files will still store empty NaN-like values for grid cells outside the subdomain, but for even faster simulations and smaller output files set the `grid -> unmask_output` option to `false`. Disabling this option causes the output files to only store values for grid cells within the subdomain. These smaller files will likely take extra processing to effectively interoperate with other models.

CHAPTER

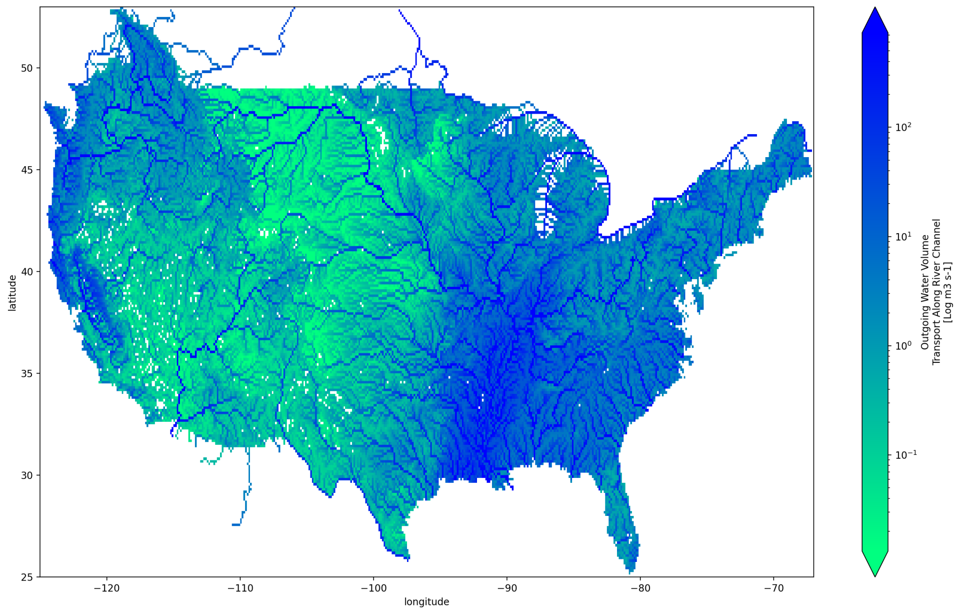
SIX

VISUALIZATION

Model instances can plot the current value of certain input and output variables (those available from Model.get_output_var_name and Model.get_input_var_names):

```
from mosartwmpy import Model
config_file = 'config.yaml'
mosart_wm = Model()
mosart_wm.initialize(config_file)
for _ in range(8):
    mosart_wm.update()

mosart_wm.plot_variable('outgoing_water_volume_transport_along_river_channel', log_
↪scale=True)
```

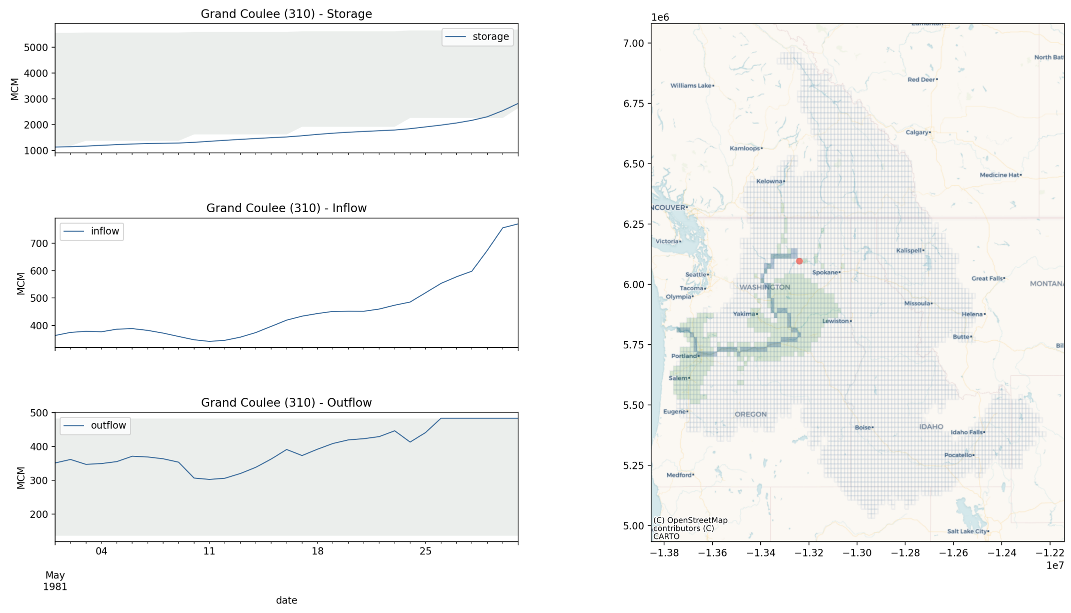


Using provided utility functions, the output of a simulation can be plotted as well.

Plot the storage, inflow, and outflow of a particular GRanD dam:

```
from mosartwmpy import Model
from mosartwmpy.plotting.plot import plot_reservoir
config_file = 'config.yaml'
mosart_wm = Model()
mosart_wm.initialize(config_file)
mosart_wm.update_until()

plot_reservoir(
    model=mosart_wm,
    grand_id=310,
    start='1981-05-01',
    end='1981-05-31',
)
```



Plot a particular output variable (as defined in `config.yaml`) over time:

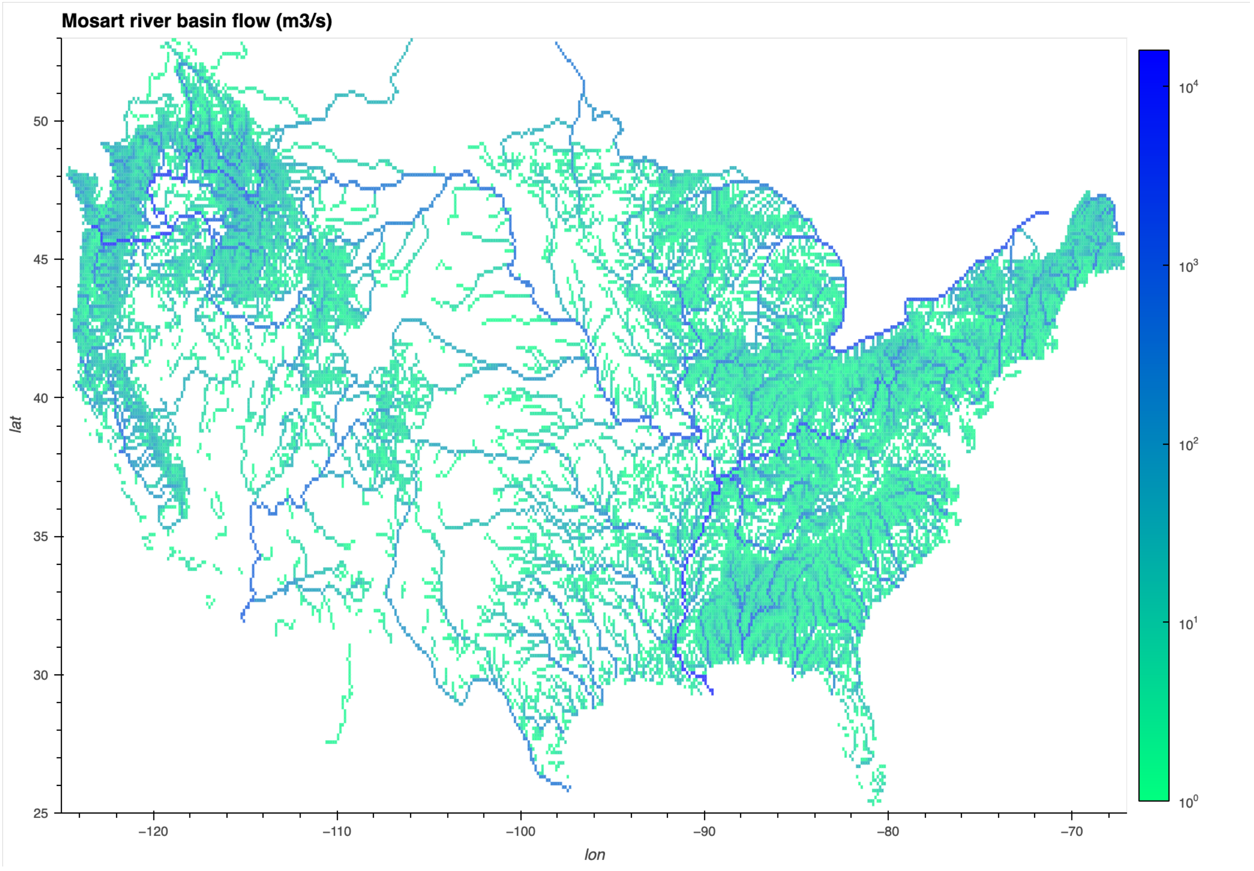
```
from mosartwmpy import Model
from mosartwmpy.plotting.plot import plot_variable
config_file = 'config.yaml'
mosart_wm = Model()
mosart_wm.initialize(config_file)
mosart_wm.update_until()

plot_variable(
    model=mosart_wm,
    variable='RIVER_DISCHARGE_OVER_LAND_LIQ',
    start='1981-05-01',
    end='1981-05-31',
    log_scale=True,
```

(continues on next page)

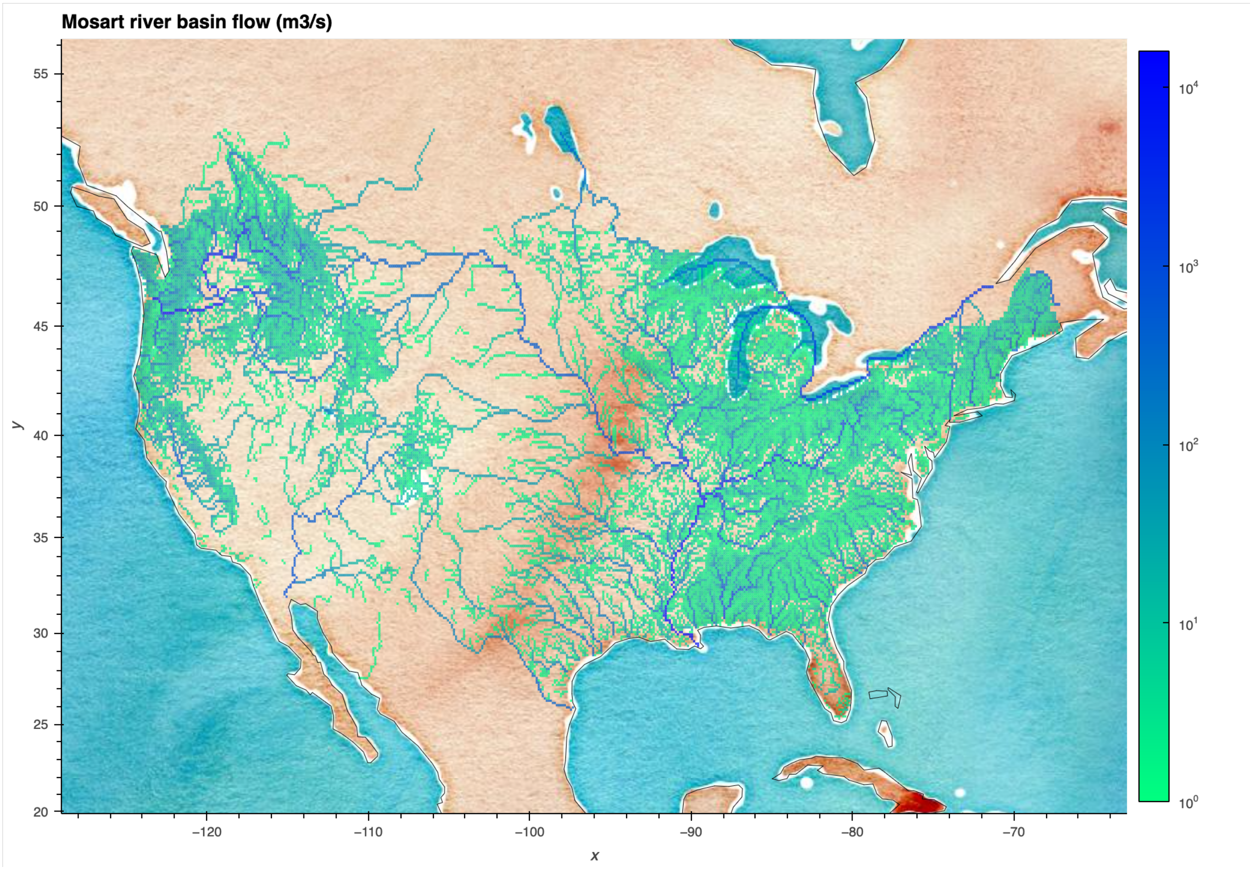
(continued from previous page)

```
    cmap='winter_r',
)
```



If `cartopy`, `scipy`, and `geoviews` are installed, tiles can be displayed along with the plot:

```
plot_variable(
    model=mosart_wm,
    variable='RIVER_DISCHARGE_OVER_LAND_LIQ',
    start='1981-05-01',
    end='1981-05-31',
    log_scale=True,
    cmap='winter_r',
    tiles='StamenWatercolor'
)
```


Mosart river basin flow (m3/s)
55
50
45
40
35
30
25
20
y
−120
−110
−100
−90
−80
−70
x
10^4
10^3
10^2
10^1
10^0

CHAPTER

SEVEN

MODEL COUPLING

A common use case for `mosartwmpy` is to run coupled with output from the Community Land Model (CLM). To see an example of how to drive `mosartwmpy` with runoff from a coupled model, check out the Jupyter notebook tutorial!

CHAPTER

EIGHT

TESTING AND VALIDATION

Before running the tests or validation, make sure to download the "sample_input" and "validation" datasets using the download utility `python -m mosartwmpy.download`.

To execute the tests, run `./test.sh` or `python -m unittest discover mosartwmpy/tests` from the repository root.

To execute the validation, run a model simulation that includes the years 1981 - 1982, note your output directory, and then run `python -m mosartwmpy.validate` from the repository root. This will ask you for the simulation output directory, think for a moment, and then open a figure with several plots representing the NMAE (Normalized Mean Absolute Error) as a percentage and the spatial sums of several key variables compared between your simulation and the validation scenario. Use these plots to assist you in determining if the changes you have made to the code have caused unintended deviation from the validation scenario. The NMAE should be 0% across time if you have caused no deviations. A non-zero NMAE indicates numerical difference between your simulation and the validation scenario. This might be caused by changes you have made to the code, or alternatively by running a simulation with different configuration or parameters (i.e. larger timestep, fewer iterations, etc). The plots of the spatial sums can assist you in determining what changed and the overall magnitude of the changes.

If you wish to merge code changes that intentionally cause significant deviation from the validation scenario, please work with the maintainers to create a new validation dataset.

CHAPTER

NINE

MOSARTWMPY TUTORIAL

This tutorial will demonstrate the basic use of `mosartwmpy` in two scenarios: * First, in standalone mode where all model inputs are provided from files * Second, a contrived example of running the model coupled with runoff input from another model

The use of restart files will also be demonstrated.

[1]:
```
from datetime import date, datetime
import numpy as np

from mosartwmpy import Model
from mosartwmpy.plotting.plot import plot_reservoir, plot_variable
from mosartwmpy.utilities.download_data import download_data
```

```
/Users/thur961/im3/mosartwmpy/venv/lib/python3.9/site-packages/pandas/compat/__init__.py:
↪124: UserWarning: Could not import the lzma module. Your installed Python is␣
↪incomplete. Attempting to use lzma compression will result in a RuntimeError.
  warnings.warn(msg)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Download and unpack the tutorial dataset, which covers May of 1981 (this may take a few minutes):

[2]:
```
download_data('tutorial')
```

```
Downloading example data from https://zenodo.org/record/5597952/files/mosartwmpy-
↪tutorial-1981-05.zip?download=1
https://zenodo.org/record/5597952/files/mosartwmpy-tutorial-1981-05.zip?download=1: 100
↪%|| 16.1M/16.1M [00:05<00:00, 3.22MB/s]
Unzipped: ./input/
Unzipped: ./input/domains/
Unzipped: ./input/domains/land.nc
Unzipped: ./input/domains/mosart.nc
Unzipped: ./input/reservoirs/
Unzipped: ./input/reservoirs/mean_monthly_reservoir_flow.parquet
Unzipped: ./input/reservoirs/mean_monthly_reservoir_demand.parquet
Unzipped: ./input/reservoirs/reservoirs.nc
Unzipped: ./input/reservoirs/dependency_database.parquet
```

(continues on next page)

(continued from previous page)

```
Unzipped: ./input/runoff/
Unzipped: ./input/runoff/runoff_1981_05.nc
Unzipped: ./input/demand/
Unzipped: ./input/demand/demand_1981_05.nc
Download and install complete.
```

Initialize the model using the provided `config.yaml`, which has the input paths properly specified for this notebook:

[3]:
```
mosart_wm = Model()
mosart_wm.initialize('./config.yaml')
```

```
Initalizing model...
Done.
```

The model is now setup to run in standalone mode, with runoff provided from file.

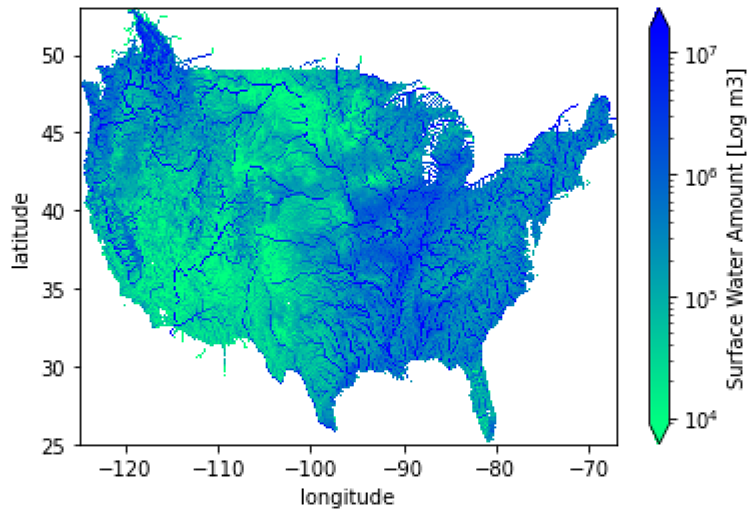
Let's run for a couple weeks in this mode (this may take a couple minutes):

[4]:
```
mosart_wm.config["simulation.end_date"] = date(1981, 5, 14)
mosart_wm.update_until(mosart_wm.get_end_time())
```

```
Beginning simulation for 1981-05-01 through 1981-05-14...
Running mosartwmpy
Simulation completed in 1 minutes and 25 seconds.
```

Let's take a look at how the river channels have formed over this week.

[5]:
```
mosart_wm.plot_variable('surface_water_amount', log_scale=True)
```



We can see the dominant river channels beginning to form! Since the initial conditions have no surface water, it can take a few weeks of model time to get to a good baseline, depending on the amount of rainfall.

Now let's pretend to run the model in coupled mode, as if runoff were being provided by another model such as CLM.

First, we'll implement dummy functions that create random runoff to take the place of the coupled model:

[6]:
```
def get_surface_runoff():
    # provide the surface runoff for each grid cell in mm/s
```

(continues on next page)

(continued from previous page)

```
    data = np.random.normal(0.001, 0.001, mosart_wm.get_grid_shape())
    return np.where(
        data < 0,
        0,
        data
    )

def get_subsurface_runoff():
    # provide the subsurface runoff for each grid cell in mm/s
    data = np.random.normal(0.00015, 0.00015, mosart_wm.get_grid_shape())
    return np.where(
        data < 0,
        0,
        data
    )
```

In coupled mode, we'll need to run the model in chunks of time that correspond to the coupling period and update the coupled variables between chunks. For this example, let's say the coupling period is one day – so we'll update the runoff variables after each day of model time. The same technique could be extended to run the coupled model between each day rather than generating synthetic data.

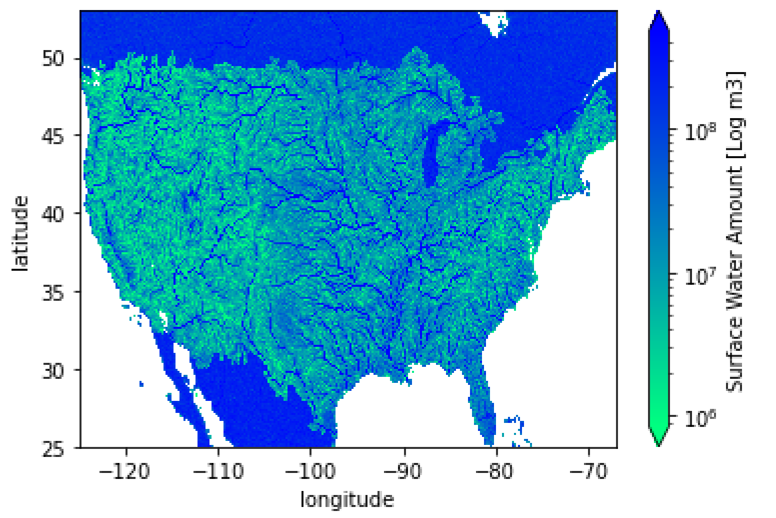
[7]:
```
# first, disable reading runoff from file
mosart_wm.config["runoff.read_from_file"] = False

# set the end date for two weeks out
mosart_wm.config["simulation.end_date"] = date(1981, 5, 28)

# for each day, set the runoff and simulate for one day
# note that the model expects the input flattened as a 1-d array
while mosart_wm.get_current_time() < mosart_wm.get_end_time():
    # set the surface runoff from a coupled model
    mosart_wm.set_value('surface_runoff_flux', get_surface_runoff().flatten())
    # set the subsurface runoff from a coupled model
    mosart_wm.set_value('subsurface_runoff_flux', get_subsurface_runoff().flatten())
    # simulate for one day
    day = date.fromtimestamp(mosart_wm.get_current_time())
    while day == date.fromtimestamp(mosart_wm.get_current_time()):
        mosart_wm.update()
```

Let's see how the random rainfall affected the river channels…

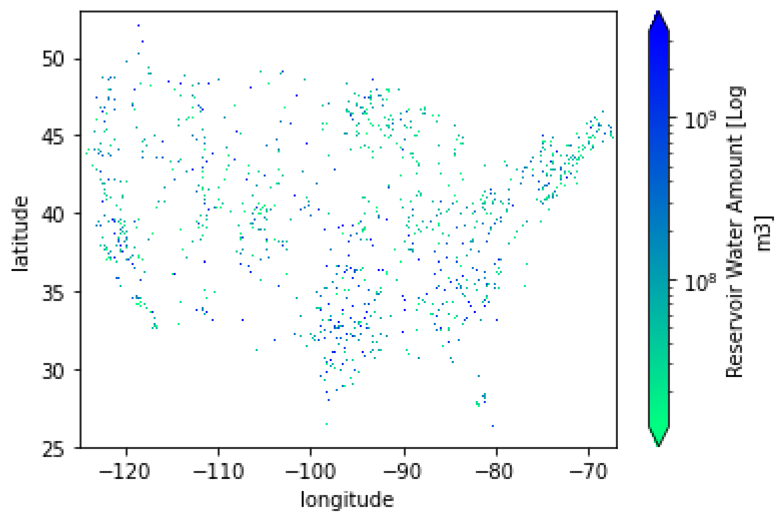
[8]:
```
mosart_wm.plot_variable('surface_water_amount', log_scale=True)
```



Oops, looks like we flooded Canada and Mexico. The river network isn't very well defined there (the tutorial only includes CONUS data for the most part) so it makes sense. Depending on the random data generation, you're probably seeing a lot more water across the CONUS too.

And how about the reservoir storage?

```
[9]: mosart_wm.plot_variable('reservoir_water_amount', log_scale=True)
```



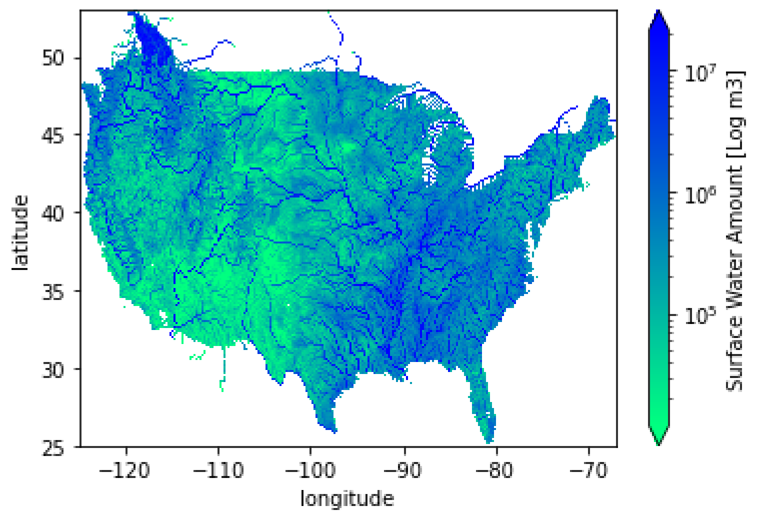
Let's try restarting the simulation after the original two weeks in May, and run the second two weeks with the runoff from the file. By default, restart files are produced at the end of each model year and also at the end of each simulation. The provided `config_with_restart.yaml` file will use the restart file from 1981-05-15. You can also use a restart file like an initial conditions file by setting the start date to whatever date you choose. But in this case we'll restart where we left off.

```
[10]: mosart_wm = Model()
mosart_wm.initialize('./config_with_restart.yaml')
mosart_wm.config["simulation.start_date"] = date(1981, 5, 15)
mosart_wm.config["simulation.end_date"] = date(1981, 5, 28)
mosart_wm.update_until(mosart_wm.get_end_time())
```

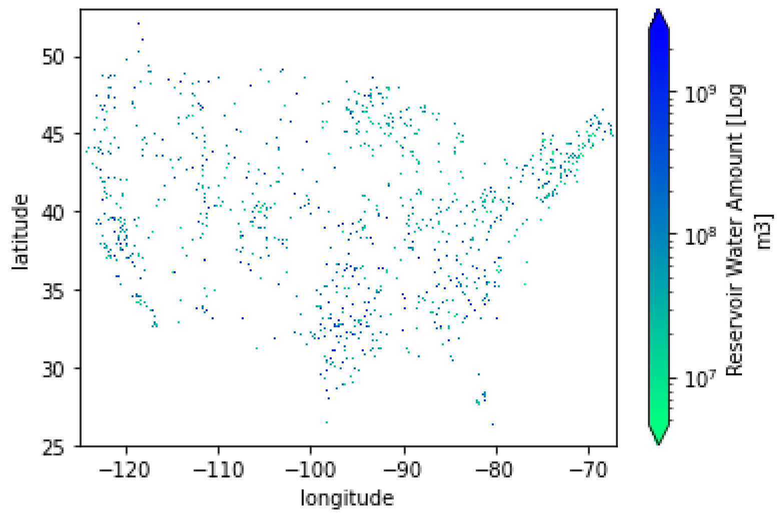
```
Initalizing model...
Loading restart file from: `./output/tutorial/restart_files/tutorial_restart_1981_05_15.
↪nc`.
Done.
Beginning simulation for 1981-05-15 through 1981-05-28...
Running mosartwmpy
Simulation completed in 1 minutes and 21 seconds.
```

Let's look at the surface water and reservoir storage again:

[11]:
```
mosart_wm.plot_variable('surface_water_amount', log_scale=True)
```



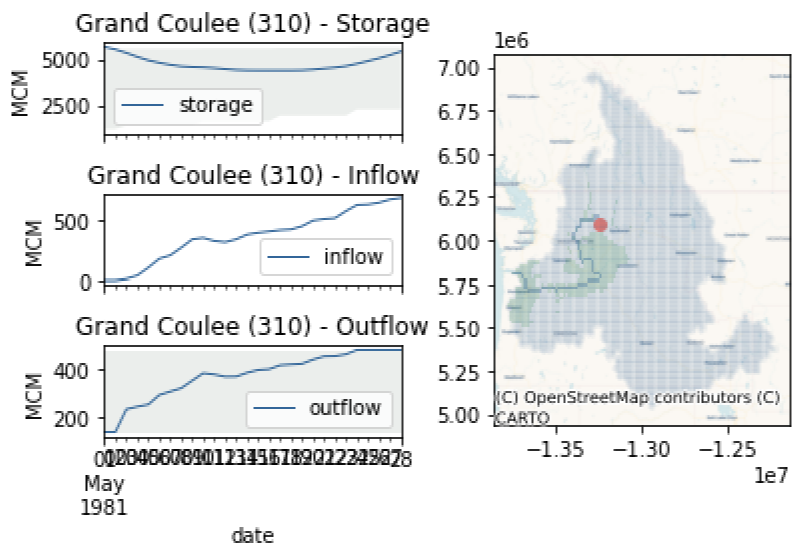
[12]:
```
mosart_wm.plot_variable('reservoir_water_amount', log_scale=True)
```



These plots make more sense than the random runoff!

`mosartwmpy` also includes visualization methods to examine the output over time. For instance, one can plot the behavior of a specific reservoir or create a timeplayer of the water deficit:

```
[13]: plot_reservoir(
          model=mosart_wm,
          grand_id=310,
          start='1981-05-01',
          end='1981-05-31',
      )
```



```
[14]: plot_variable(
          model=mosart_wm,
          variable='WRM_DEFICIT',
          start='1981-05-01',
          end='1981-05-31',
          log_scale=True,
          cmap='autumn_r',
      )

      Launching server at http://localhost:58014
```

This concludes the `mosartwmpy` tutorial. Feel free to open issues with your feedback or start topics on the discussion board!

CHAPTER

TEN

PYTHON VIRTUAL ENVIRONMENTS

Maintaining different versions of Python can be a chore. Thankfully, there are many tools for managing Python environments; here are a few recommendations:

- PyCharm IDE – great for developing code in Python, and can automatically create virtual environments for a codebase by detecting versions and dependencies from the `setup.py` or `setup.cfg`.
- Conda package manager – a Python package manager focused on scientific computing that can also manage virtual environments.
- pyenv CLI – a shell based tool for installing and switching between different versions of Python and dependencies. I will give a brief tutorial of using `pyenv` below, but recognize that the instructions may change over time so the `pyenv` documentation is the best place to look.

To create a Python 3.9 virtual environment, try the following steps:

- Install pyenv:
  - if on Mac, use brew: brew install pyenv
  - if on a linux system, try pyenv-installer
  - if on Windows, try pyenv-win
- Install Python 3.9:
  - in a shell, run `pyenv install 3.9.1`
- Activate Python 3.9 in the current shell
  - in the shell, run `pyenv shell 3.9.1`
- Proceed with the install of mosartwmpy:
  - in the same shell, run `pip install mosartwmpy`
- Now you can interact with `mosartwmpy` in this current shell session
  - if you start a new shell session you will need to run `pyenv shell 3.9.1` again before proceeding
  - this new shell session should maintain all previously pip installed modules for Python 3.9.1

CHAPTER

ELEVEN

MOSARTWMPY PYTHON API

11.1 mosartwmpy.model module

`class mosartwmpy.model.`**`Model`**

Bases: `bmipy.bmi.Bmi`

The mosartwmpy basic model interface.

Parameters **`Bmi`** (*`Bmi`*) – The Basic Model Interface class

Returns A BMI instance of the MOSART-WM model.

Return type *Model*

`finalize``()` → None

Perform tear-down tasks for the model.

Perform all tasks that take place after exiting the model's time loop. This typically includes deallocating memory, closing files and printing reports.

`get_component_name``()` → str

Name of the component.

Returns The name of the component.

Return type str

`get_current_time``()` → float

Current time of the model.

Returns The current model time.

Return type float

`get_end_time``()` → float

End time of the model.

Returns The maximum model time.

Return type float

`get_grid_edge_count``(`*grid: int = 0*`)` → int

Get the number of edges in the grid.

Parameters **`grid`** (*`int`*) – A grid identifier.

Returns The total number of grid edges.

Return type int

`get_grid_edge_nodes`(*grid: int = 0*, *edge_nodes: Optional[numpy.ndarray] = None*) → numpy.ndarray
Get the edge-node connectivity.

Parameters

- **grid** (*int*) – A grid identifier.
- **edge_nodes** (ndarray of int, shape *(2 x nnodes,)*) – A numpy array to place the edge-node connectivity. For each edge, connectivity is given as node at edge tail, followed by node at edge head.

Returns The input numpy array that holds the edge-node connectivity.

Return type ndarray of int

`get_grid_face_count`(*grid: int = 0*) → int
Get the number of faces in the grid.

Parameters **grid** (*int*) – A grid identifier.

Returns The total number of grid faces.

Return type int

`get_grid_face_edges`(*grid: int = 0*, *face_edges: Optional[numpy.ndarray] = None*) → numpy.ndarray
Get the face-edge connectivity.

Parameters

- **grid** (*int*) – A grid identifier.
- **face_edges** (*ndarray of int*) – A numpy array to place the face-edge connectivity.

Returns The input numpy array that holds the face-edge connectivity.

Return type ndarray of int

`get_grid_face_nodes`(*grid: int = 0*, *face_nodes: Optional[numpy.ndarray] = None*) → numpy.ndarray
Get the face-node connectivity.

Parameters

- **grid** (*int*) – A grid identifier.
- **face_nodes** (*ndarray of int*) – A numpy array to place the face-node connectivity. For each face, the nodes (listed in a counter-clockwise direction) that form the boundary of the face.

Returns The input numpy array that holds the face-node connectivity.

Return type ndarray of int

`get_grid_node_count`(*grid: int = 0*) → int
Get the number of nodes in the grid.

Parameters **grid** (*int*) – A grid identifier.

Returns The total number of grid nodes.

Return type int

`get_grid_nodes_per_face`(*grid: int = 0*, *nodes_per_face: Optional[numpy.ndarray] = None*) → numpy.ndarray
Get the number of nodes for each face.

Parameters

- **grid** (*int*) – A grid identifier.

- **nodes_per_face** (ndarray of int, shape *(nfaces,)*) – A numpy array to place the number of edges per face.

Returns The input numpy array that holds the number of nodes per edge.

Return type ndarray of int

get_grid_origin(*grid: int = 0*, *origin: numpy.ndarray = array([0.0, 6.91389664e-310])*) → numpy.ndarray

Get coordinates for the lower-left corner of the computational grid.

Parameters

- **grid** (*int*) – A grid identifier.
- **origin** (ndarray of float, shape *(ndim,)*) – A numpy array to hold the coordinates of the lower-left corner of the grid.

Returns The input numpy array that holds the coordinates of the grid's lower-left corner.

Return type ndarray of float

get_grid_rank(*grid: int = 0*) → int

Get number of dimensions of the computational grid.

Parameters **grid** (*int*) – A grid identifier.

Returns Rank of the grid.

Return type int

get_grid_shape(*grid: int = 0*, *shape: numpy.ndarray = array([0, 94155524011368])*) → numpy.ndarray

Get dimensions of the computational grid.

Parameters

- **grid** (*int*) – A grid identifier.
- **shape** (ndarray of int, shape *(ndim,)*) – A numpy array into which to place the shape of the grid.

Returns The input numpy array that holds the grid's shape.

Return type ndarray of int

get_grid_size(*grid: int = 0*) → int

Get the total number of elements in the computational grid.

Parameters **grid** (*int*) – A grid identifier.

Returns Size of the grid.

Return type int

get_grid_spacing(*grid: int = 0*, *spacing: numpy.ndarray = array([6.91389664e-310, 4.65190098e-310])*) → numpy.ndarray

Get distance between nodes of the computational grid.

Parameters

- **grid** (*int*) – A grid identifier.
- **spacing** (ndarray of float, shape *(ndim,)*) – A numpy array to hold the spacing between grid rows and columns.

Returns The input numpy array that holds the grid's spacing.

Return type ndarray of float

`get_grid_type`(*grid: int = 0*) → str
Get the grid type as a string.

Parameters `grid` (*int*) – A grid identifier.

Returns Type of grid as a string.

Return type str

`get_grid_x`(*grid: int = 0*, *x: Optional[numpy.ndarray] = None*) → numpy.ndarray
Get coordinates of grid nodes in the x direction.

Parameters

- **grid** (*int*) – A grid identifier.
- **x** (ndarray of float, shape *(nrows,)*) – A numpy array to hold the x-coordinates of the grid node columns.

Returns The input numpy array that holds the grid's column x-coordinates.

Return type ndarray of float

`get_grid_y`(*grid: int = 0*, *y: Optional[numpy.ndarray] = None*) → numpy.ndarray
Get coordinates of grid nodes in the y direction.

Parameters

- **grid** (*int*) – A grid identifier.
- **y** (ndarray of float, shape *(ncols,)*) – A numpy array to hold the y-coordinates of the grid node rows.

Returns The input numpy array that holds the grid's row y-coordinates.

Return type ndarray of float

`get_grid_z`(*grid: int = 0*, *z: Optional[numpy.ndarray] = None*) → numpy.ndarray
Get coordinates of grid nodes in the z direction.

Parameters

- **grid** (*int*) – A grid identifier.
- **z** (ndarray of float, shape *(nlayers,)*) – A numpy array to hold the z-coordinates of the grid nodes layers.

Returns The input numpy array that holds the grid's layer z-coordinates.

Return type ndarray of float

`get_input_item_count`() → int
Count of a model's input variables.

Returns The number of input variables.

Return type int

`get_input_var_names`() → Tuple[str]
List of a model's input variables.

Input variable names must be CSDMS Standard Names, also known as *long variable names*.

Returns The input variables for the model.

Return type list of str

Notes

Standard Names enable the CSDMS framework to determine whether an input variable in one model is equivalent to, or compatible with, an output variable in another model. This allows the framework to automatically connect components.

Standard Names do not have to be used within the model.

`get_output_item_count()` → int
Count of a model's output variables.

Returns The number of output variables.

Return type int

`get_output_var_names()` → Tuple[str]
List of a model's output variables.

Output variable names must be CSDMS Standard Names, also known as *long variable names*.

Returns The output variables for the model.

Return type list of str

`get_start_time()` → float
Start time of the model.

Model times should be of type float.

Returns The model start time.

Return type float

`get_time_step()` → float
Current time step of the model.

The model time step should be of type float.

Returns The time step used in model.

Return type float

`get_time_units()` → str
Time units of the model.

Returns The model time unit; e.g., *days* or *s*.

Return type float

Notes

CSDMS uses the UDUNITS standard from Unidata.

`get_value`(*name: str*, *dest: numpy.ndarray*) → int
Get a copy of values of the given variable.

This is a getter for the model, used to access the model's current state. It returns a *copy* of a model variable, with the return type, size and rank dependent on the variable.

Parameters

- **name** (*str*) – An input or output variable name, a CSDMS Standard Name.
- **dest** (*ndarray*) – A numpy array into which to place the values.

Returns The same numpy array that was passed as an input buffer.

Return type ndarray

`get_value_at_indices`(*name: str*, *dest: numpy.ndarray*, *inds: numpy.ndarray*) → int
Get values at particular indices.

Parameters

- `name` (*str*) – An input or output variable name, a CSDMS Standard Name.
- `dest` (*ndarray*) – A numpy array into which to place the values.
- `indices` (*array_like*) – The indices into the variable array.

Returns Value of the model variable at the given location.

Return type array_like

`get_value_ptr`(*name: str*) → numpy.ndarray
Get a reference to values of the given variable.

This is a getter for the model, used to access the model's current state. It returns a reference to a model variable, with the return type, size and rank dependent on the variable.

Parameters `name` (*str*) – An input or output variable name, a CSDMS Standard Name.

Returns A reference to a model variable.

Return type array_like

`get_var_grid`(*name: str*) → int
Get grid identifier for the given variable.

Parameters `name` (*str*) – An input or output variable name, a CSDMS Standard Name.

Returns The grid identifier.

Return type int

`get_var_itemsize`(*name: str*) → int
Get memory use for each array element in bytes.

Parameters `name` (*str*) – An input or output variable name, a CSDMS Standard Name.

Returns Item size in bytes.

Return type int

`get_var_location`(*name: str*) → str
Get the grid element type that the a given variable is defined on.

The grid topology can be composed of *nodes*, *edges*, and *faces*.

node A point that has a coordinate pair or triplet: the most basic element of the topology.

edge A line or curve bounded by two *nodes*.

face A plane or surface enclosed by a set of edges. In a 2D horizontal application one may consider the word "polygon", but in the hierarchy of elements the word "face" is most common.

Parameters `name` (*str*) – An input or output variable name, a CSDMS Standard Name.

Returns The grid location on which the variable is defined. Must be one of *"node"*, *"edge"*, or *"face"*.

Return type str

Notes

CSDMS uses the ugrid conventions to define unstructured grids.

`get_var_nbytes`(*name: str*) → int
Get size, in bytes, of the given variable.

Parameters `name` (*str*) – An input or output variable name, a CSDMS Standard Name.

Returns The size of the variable, counted in bytes.

Return type int

`get_var_type`(*name: str*) → str
Get data type of the given variable.

Parameters `name` (*str*) – An input or output variable name, a CSDMS Standard Name.

Returns The Python variable type; e.g., `str`, `int`, `float`.

Return type str

`get_var_units`(*name: str*) → str
Get units of the given variable.

Standard unit names, in lower case, should be used, such as `meters` or `seconds`. Standard abbreviations, like `m` for meters, are also supported. For variables with compound units, each unit name is separated by a single space, with exponents other than 1 placed immediately after the name, as in `m s-1` for velocity, `W m-2` for an energy flux, or `km2` for an area.

Parameters `name` (*str*) – An input or output variable name, a CSDMS Standard Name.

Returns The variable units.

Return type str

Notes

CSDMS uses the UDUNITS standard from Unidata.

`initialize`(*config_file_path: str = './config.yaml'*, *grid: Optional[mosartwmpy.grid.grid.Grid] = None*, *state: Optional[mosartwmpy.state.state.State] = None*) → None
Perform startup tasks for the model.

Perform all tasks that take place before entering the model's time loop, including opening files and initializing the model state. Model inputs are read from a text-based configuration file, specified by *filename*.

Parameters `config_file` (*str, optional*) – The path to the model configuration file.

Notes

Models should be refactored, if necessary, to use a configuration file. CSDMS does not impose any constraint on how configuration files are formatted, although YAML is recommended. A template of a model's configuration file with placeholder values is used by the BMI.

`plot_variable`(*variable: str*, *log_scale: bool = False*, *show: bool = True*)
Display a colormap of a spatial variable at the current timestep.

`set_value`(*name: str*, *src: numpy.ndarray*) → int
Specify a new value for a model variable.

This is the setter for the model, used to change the model's current state. It accepts, through *src*, a new value for a model variable, with the type, size and rank of *src* dependent on the variable.

Parameters

- **`var_name`** (*`str`*) – An input or output variable name, a CSDMS Standard Name.
- **`src`** (*`array_like`*) – The new value for the specified variable.

`set_value_at_indices`(*name: str*, *inds: numpy.ndarray*, *src: numpy.ndarray*) → int

Specify a new value for a model variable at particular indices.

Parameters

- **`var_name`** (*`str`*) – An input or output variable name, a CSDMS Standard Name.
- **`indices`** (*`array_like`*) – The indices into the variable array.
- **`src`** (*`array_like`*) – The new value for the specified variable.

`unmask`(*vector: numpy.ndarray*) → numpy.ndarray

`update`() → None

Advance model state by one time step.

Perform all tasks that take place within one pass through the model's time loop. This typically includes incrementing all of the model's state variables. If the model's state variables don't change in time, then they can be computed by the *`initialize()`* method and this method can return with no action.

`update_until`(*time: Optional[float] = None*) → None

Advance model state until the given time.

Parameters **`time`** (*`float`*) – A model time later than the current model time.

PYTHON MODULE INDEX

INDEX