OMUSE Documentation

the OMUSE team

Jul 01, 2022

Contents:

CHAPTER 1

Installing OMUSE

Install required programs and libraries:

- make
- cmake
- python 3
- gcc
- gfortran
- mpi
- netCDF including Fortran bindings
- git
- mercurial

Set up and activate a virtual Python environment:

```
python3 -m venv omuse_env
source omuse_env/bin/activate
```

Get the OMUSE source code, install its Python dependencies and set up a development build of OMUSE, where the codes will be built in place:

```
git clone https://github.com/omuse-geoscience/omuse/
cd omuse/
pip install -e .
export DOWNLOAD_CODES=1
```

Build codes, select the ones needed:

```
python setup.py build_code --code-name dales   --inplace
python setup.py build_code --code-name cdo     --inplace
python setup.py build_code --code-name qgcm    --inplace
```

(continues on next page)

(continued from previous page)

```
python setup.py build_code --code-name qgmodel --inplace
python setup.py build_code --code-name swan    --inplace
python setup.py build_code --code-name pop     --inplace
```

or try to build all of them:

```
python setup.py develop_build
```

Install Jupyter in the virtual environment, and make the virtual environment's Python available as a kernel in Jupyter

python -m pip install ipykernel matplotlib python -m ipykernel install –user –name=omuse-env

Alternatively, see *Singularity image* for instructions for setting up and using a Singularity container with OMUSE and Jupyter.

1.1 Code versions

For DALES, there are additional options controlling which version of the code is used: setting *DOWNLOAD_CODES=1* performs a shallow checkout of a single tag, while *DOWNLOAD_CODES="all"* clones the whole DALES git repository, which is useful for development. The environment variable *DALES_GIT_TAG* can be used to control which branch or version tag to check out. By default the variable points to a version tag in the DALES repository, which is tested to work with the current OMUSE.

CHAPTER 2

DALES interface

This is the OMUSE interface to DALES, the Dutch Atmospheric Large-Eddy Simulation.

Example:

```
from omuse.community.dales.interface import Dales
from omuse.units import units

d = Dales(workdir='dales-workdir', channel_type='sockets', number_of_workers=1)

# set parameters
d.parameters_DYNAMICS.iadv_mom = 6  # 6th order advection for momentum
d.parameters_DYNAMICS.iadv_thl = 5  # 5th order advection for scalars, less␣
↪overshoots than 6th order
d.parameters_DYNAMICS.iadv_qt = 5
d.parameters_DYNAMICS.iadv_tke = 5

# access model variables
d.fields[:, :, :].U = 0 | units.m / units.s

# evolve the model
d.evolve_model(120 | units.s, exactEnd=True)
```

OMUSE models use variables with units, see *Units in OMUSE*.

The parameters are doumented in the DALES documentation: options. Setting parameters is possible before the model is time stepped or the model variables have been accessed, but not after.

2.1 Variable grids

various variables of the DALES model can be accessed from the OMUSE interface. The variables are grouped in grids, according to their dimensionality. For example, the U-velocity component in the model component in the model can be accessed as `d.fields[i,j,k].U`.

The following grids are defined:

Grid	Description
fields	3D grid
profiles	horizontal averages of the 3D grid (vertical profiles)
nudging_profiles	vertical profiles of U, V, THL, QT to nudge the model towards.
forcing_profiles	external forcings of U, V, THL, QT - vertical profiles
scalars	surface fluxes and roughness lengths, scalars.
surface_fields	2D surface fields, e.g. liquid water path.

The grids contain the following variables:

Grid	Variables
fields	U, V, W, THL, QT, QL, QL_ice, QR, E12, T, pi, rswd, rswdir, rswdif,
	rswu, rlwd, rlwu, rswdcs, rswucs, rlwdcs, rlwucs
profiles	U, V, W, THL, QT, QL, QL_ice, QR, E12, T, P, rho
nudging_profiles	U, V, THL, QT
forcing_profiles	U, V, THL, QT
scalars	wt, wq, z0m, z0h, Ps, QR
surface_fields	LWP, RWP, TWP, ustar, z0m, z0h, tskin, qskin, LE, H, obl

Description of the variables:

Variable	Unit	Description
U,V,W	m/s	velocity components in x, y, and z directions
THL	K	liquid water potential temperature
QT	kg/kg	specific total humidity (i.e. vapor and cloud condensate but excluding precipitation)
QL	kg/kg	specific cloud condensate
QL_ice	kg/kg	specific cloud condensate in the form of ice
QR	kg/kg	precipitation
E12	m/s	sqrt(subgrid-scale turbulent kinetic energy)
T	K	temperature
P	Pa	pressure
Ps	Pa	surface pressure
rho	kg/m^3	density
pi	Pa	modified air pressure
rswu,rswd	W/m^2	up-/down-welling short-wave radiative flux
rlwu,rlwd	W/m^2	up-/down-welling long-wave radiative flux
r{s,l}w{u,d}cs	W/m^2	up/down-welling long/short-wave radiative flux for clear sky
rswdir	W/m^2	downwelling shortwave direct radiative flux
rswdif	W/m^2	downwelling shortwave diffusive radiative flux
wt	K m/s	surface flux of heat
wq	m/s	surface flux of humidity
z0m, z0h	m	surface roughness length
ustar	m/s	friction velocity
LWP, RWP, TWP	kg/m^2	liquid-, rain-, total water path
tskin	K	skin temperature
qskin	kg/kg	skin humidity
H, LE	W/m^2	sensible and latent heat fluxes
obl	m	Obukhov length

The forcing and nudging profiles, and the 3D grids for prognostic variables can be read and written. Diagnosed quantities, horizontal averages, and water paths can only be read.

Note: the indexing brackets can also be placed on the variable name instead of on the grid name, e.g. `d.fields.U[:, :, :]` vs `d.fields[:, :, :].U`. Avoid using the first form, since assigning new values to the grid this way does not work.

2.2 Example scripts

Some examples of using Dales with OMUSE are bundled included in omuse/src/omuse/community/dales/example/ .

2.2.1 bubble.py

A Dales experiment with a warm air bubble. Shows model initialization, setting model parameters, setting initial conditions, evolving the model, and retreiving the state.

2.2.2 async.py

Shows how to use OMUSE's asynchronous function calls, to let several model instances run concurrently. See also *Asynchronous calls*.

class omuse.community.dales.interface.**Dales**(***options*)

OMUSE Dales Interface.

Parameters

- **workdir** (*str, optional*) – Working directory for DALES. Output files are placed here. If *workdir* doesn't exist, and either inputdir or case is given, input files are copied from the provided directory to workdir. If *workdir* doesn't exist, and no case or input files are provided, built-in defaults are used.
- **exp** (*int, optional*) – Experiment number, used to number input files, e.g. prof.inp.001 .
- **inputdir** (*str, optional*) – Directory for input files, copied to workdir.
- **case** (*str, optional*) – Specify one of the cases bundled with DALES. Valid names include 'bomex', 'rico', 'atex', 'fog'. Input files are copied to workdir.
- **z** (*numpy array, optional*) – Override the vertical discretization with an array of increasing heights. The array is assumed to have a length unit attached to it.
- **interpolator** (*function handle, optional*) – In case of a user-specified vertical discretization, this function determines the interpolation method used to obtain the initial profiles at the desired resolution. Should be a function f(str,z_new,z_old,y) where the first argument denotes the variable, the second and third resp. the new and old z-axes and the latter the profile values on the opriginal axis, returning an array of values on the new axis. Default = None, meaning linear interpolation.
- **number_of_workers** (*int, optional*) – Number of MPI tasks to use. General OMUSE option. Default = 1.
- **channel_type** (*str, optional*) – Communication channel between Python and DALES worker processes. Options 'mpi' or 'sockets'. General OMUSE option.
- **redirection** (*str, optional*) – Options for re-directing stdout and stderr of the DALES process. General OMUSE option. 'none' for no redirection, 'file' for redirection to files.

- **redirect_stdout_file** (*str, optional.*) – File name for redirection of stdout, see above. General OMUSE option.
- **redirect_stderr_file** (*str, optional.*) – File name for redirection of stderr, see above. General OMUSE option.

get_dx()
Dales grid cell size in x-direction

Returns Grid resolution (in m) along U-direction

Return type float

get_dy()
Dales grid cell size in y-direction

Returns Grid resolution (in m) along V-direction

Return type float

get_field(*field*, *imin=1*, *imax=None*, *jmin=1*, *jmax=None*, *kmin=1*, *kmax=None*, ***kwargs*)
Dales volume field retrieval method

Parameters

- **field** (*str*) – Variable shortname: either LWP, RWP, TWP, U, V, W, THL, T, QT, QL, Qsat
- **imin** (*integer, optional*) – Lower one-based x-bound of data block
- **imax** (*integer, optional*) – Upper one-based x-bound of data block
- **jmin** (*integer, optional*) – Lower one-based y-bound of data block
- **jmax** (*integer, optional*) – Upper one-based y-bound of data block
- **kmin** (*integer, optional*) – Lower one-based z-bound of data block
- **kmax** (*integer, optional*) – Upper one-based z-bound of data block
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns 3D block containing variable values

Return type numpy.array

get_itot()
Dales number of grid cells in x-direction

Returns Number of grid cells along U-direction

Return type int

get_jtot()
Dales number of grid cells in y-direction

Returns Number of grid cells along V-direction

Return type int

get_ktot()
Dales number of grid cells in z-direction

Returns Number of vertical layers

Return type int

get_presf(*k=None*, ***kwargs*)
Dales full level mean pressure retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Full level mean pressure profile

Return type numpy.array

get_presh(*k=None*, ***kwargs*)
Dales half level mean pressure retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Half level mean pressure profile

Return type numpy.array

get_profile(*field*, ***kwargs*)
Dales generic profile retrieval method

Parameters

- **field** (*str*) – Variable shortname: either U, V, W, THL, T, QT, QL, E12
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns 1D array containing mean vertical profile values

Return type numpy.array

get_profile_E12(*k=None*, ***kwargs*)
Dales turbulence kinetic energy profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Turbulence kinetic energy vertical profile

Return type numpy.array

get_profile_QL(*k=None*, ***kwargs*)
Dales liquid water content profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Liquid water content vertical profile

Return type numpy.array

get_profile_QL_ice(*k=None*, ***kwargs*)
Dales ice water content profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Ice water content vertical profile

Return type numpy.array

get_profile_QR(*k=None*, ***kwargs*)
Dales rain water content profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Rain water content vertical profile

Return type numpy.array

get_profile_QT(*k=None*, ***kwargs*)
Dales total humidity profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Total humidity vertical profile

Return type numpy.array

get_profile_T(*k=None*, ***kwargs*)
Dales temperature profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Temperature vertical profile

Return type numpy.array

get_profile_THL(*k=None*, ***kwargs*)
Dales liquid water virtual temperature profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Liquid water virtual temperature vertical profile

Return type numpy.array

get_profile_U(*k=None*, ***kwargs*)
Dales eastward wind profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Eastward vertical wind profile

Return type numpy.array

get_profile_V(*k=None*, ***kwargs*)
Dales northward wind profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Northward vertical wind profile

Return type numpy.array

get_profile_W(*k=None*, ***kwargs*)
Dales upward wind profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Upward vertical wind profile

Return type numpy.array

get_rhobf(*k=None*, ***kwargs*)
Dales base density profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Full level density base profile

Return type numpy.array

get_rhof(*k=None*, ***kwargs*)
Dales mean density profile retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.

- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Full level mean density profile

Return type numpy.array

get_xsize()
Dales domain extent in x-direction

Returns Domain length (in m) along U-direction

Return type float

get_ysize()
Dales domain extent in y-direction

Returns Domain length (in m) along V-direction

Return type float

get_zf(*k=None*, ***kwargs*)
Dales full level heights retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Full level heights

Return type numpy.array

get_zh(*k=None*, ***kwargs*)
Dales half level heights retrieval method

Parameters

- **k** (*integer array, optional*) – Restrict profile to this set of vertical indices.
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

Returns Half level heights

Return type numpy.array

set_field(*field*, *a*, *imin=1*, *jmin=1*, *kmin=1*, ***kwargs*)
Dales volume field insertion method

Parameters

- **field** (*str*) – prognostic variable shortname: either U, V, W, THL, QT
- **a** (*numpy.array*) – Block of values for substitution in state
- **imin** (*integer, optional*) – Lower one-based x-bound of data block
- **jmin** (*integer, optional*) – Lower one-based y-bound of data block
- **kmin** (*integer, optional*) – Lower one-based z-bound of data block
- **async** (*boolean, optional*) – Execute function asynchronously, return request object

2.3 Links and references

- The official DALES git repository
- DALES manual
- DALES namelist options
- Dales model description paper: Formulation of the Dutch Atmospheric Large-Eddy Simulation (DALES) and overview of its applications, T. Heus et al, Geosci. Model Dev., 3, 415-444, 2010

CHAPTER 3

ERA5 interface

Using the OMUSE interface to ERA5 you can access the CDS ERA5 dataset in a code-like manner. The `ERA5` class has a model interface with a `evolve_model` method which handles the download and caching of ERA5 data in a transparent fashion.

3.1 Variables

Currently the ERA5 interface exposes single level variables from ERA5. Variables are available on standard OMUSE `grid` object.

3.2 Prerequisites

`ERA5` needs the CDSAPI installed as well as a valid access key in ‘‘ ~/.cdsapirc‘‘

3.3 Example

example:

```
from omuse.units import units
from omuse.community.era5.interface import ERA5

e=ERA5(variables=["2m_temperature", "total_precipitation"],
        nwse_boundingbox=[70, -15, 40, 15]| units.deg)

print("starting date:", e.start_datetime)
print(e.grid) # note grid has prepended the names with _ (because ERA5 variable names␣
↪are not always valid python var names)
```

(continues on next page)

(continued from previous page)

```
e.evolve_model(128 | units.hour)
val=e.grid._2m_temperature.value_in(units.K)
```

class omuse.community.era5.interface.**ERA5**(*maxcache=None, cachedir='./__era5_cache', start_datetime=datetime.datetime(1979, 1, 2, 0, 0), variables=[], grid_resolution=None, nwse_boundingbox=None, invariate_variables=['land_sea_mask'], download_timespan='day'*)

ERA5 interface

Parameters

- **cachedir** (*str, optional*) – directory used for storing cache files, default: "./__era5_cache"
- **start_datetime** (*datetime object, optional*) – starting date of model. Must be datatime object, default: datetime.datetime(1979,1,2)
- **variables** (*list of str, optional*) – variables to be retrieved. Must be list of valid strings. default: []
- **grid_resolution** (*0.25, 0.5 or 1.0 | units.deg, optional*) – Resolution of the retrieved grid. default: 0.25 | units.deg
- **nwse_boundingbox** (*None or [North,West,South,East], optional*) – Bounding box of grid, will be [90,-180,-90,180] | units.deg (whole globe) if None. default: None
- **invariate_variables** (*list of str, optional*) – invariate variables to retrieve. Will be included on the grid (but not updated). default: ["land_sea_mask"]
- **download_timespan** (*None, "day" or "month", optional*) – timespan of downloaded files. Longer timespans result in fewer but bigger downloads, default: "day"

CHAPTER 4

Asynchronous calls

Asynchronous function calls and requests is a mechanism in OMUSE to let multiple models or code instances do work concurrently.

Any method call on an OMUSE object can be made asynchronous, by appending `.asynchronous` to the method's name. When an asynchronous call is made to a code, the Python script continues to run while the call is being made. The Python script can then perform computations or calls to other codes in OMUSE. Subsequent calls to the same code instance can be made, but they will be performed in sequence.

An asynchronous OMUSE function call returns immediately, i.e. it does not wait for the worker code to finish the function. Instead of the normal return value, the asynchronous function returns a request object, which can be used to access the result of the function call later. The request can also be added to a request pool, which manages several concurrent requests.

```
request1 = d1.evolve_model.asynchronous(target_time, exactEnd=True)
# ... do something else ...
print(request1.result())
```

4.1 Example

Running two models simultaneously:

```
from omuse.community.dales.interface import Dales
from omuse.units import units
from amuse.rfi.async_request import AsyncRequestsPool

# create Dales objects
d1 = Dales(workdir='dales1', channel_type='sockets', number_of_workers=1, case='bomex
↪')
d2 = Dales(workdir='dales2', channel_type='sockets', number_of_workers=1, case='bomex
↪')

# create a pool for managing asynchronous requests
```

(continues on next page)

(continued from previous page)

```
pool = AsyncRequestsPool()

# add requests to the two codes to the pool
request1 = d1.evolve_model.asynchronous(target_time, exactEnd=True)
pool.add_request(request1)

request2 = d2.evolve_model.asynchronous(target_time, exactEnd=True)
pool.add_request(request2)

# wait for the requests to finish
pool.waitall()
```

Asynchronous variable access:

```
# setting grid data
# normal synchronous call
d1.fields[:,:,3:6].U = 1 | units.m / units.s

# asynchronous call
d2.fields[:,:,3:6].request.U = 1 | units.m / units.s

# getting grid data
# synchronous call, returns an array
uprofile = d1.fields[5,5,1:10].U

# asynchronous call, returns a request.
request = d2.fields[5,5,1:10].request.U
uprofile = request.result() # retrieve result. Implicit wait for the request to finish
```

4.2 Caveats

- Mixing asynchronous and synchronous calls produces correct results, but has consequences for the sequencing of the work: Performing a normal, synchronous call to a code after one or more asynchronous calls, causes an implicit wait for the asynchronous calls to complete.
- Accessing `result()` on a request causes a wait for the request to complete.
- When making several calls to the same code instance, the first call begins immediately. The second call begins only when the code is waited on, not automatically when the first call completes.

CHAPTER 5

Units in OMUSE

OMUSE code interfaces use quantities with units. This has the advantages that the units of any quantitiy is explicitly specified, and that automatic conversion can be done between different units.

Unit example:

```
from omuse.units import units

# units are attached to a number with the | operator:
velocity = 3 | units.m / units.s
mass = 0.002 | units.kg

print('velocity', velocity)

# a quantity can be separated into a number and a unit
print('number:', velocity.number, 'unit:', velocity.unit)
print()

print('mass', mass)

# get value in a different unit
print(mass.value_in(units.g), 'g')
print()

# arithmetic on quantities:
print('momentum:', mass * velocity)
```

output:

```
velocity 3 m / s
number: 3 unit: m / s

mass 0.002 kg
2.0 g

momentum: 0.006 m * kg * s**-1
```

CHAPTER 6

Singularity image

A Singularity recipe is included with OMUSE. When building it, the result is a Singularity image which is a portable way to test OMUSE and the included models.

The image includes Jupyter for interactive Python notebooks and the following OMUSE models:

- cdo
- dales
- qgcm (hogg2006,ocean_only,hogg2006_20km)
- qgmodel
- swan

Building the image:

```
sudo singularity build omuse.img Singularity
```

Create a directory which the container can use at runtime for storing notebooks:

```
mkdir run
```

Launch the container to start Jupyter server inside:

```
singularity run --contain -B examples:/opt/notebooks,run:/run/user omuse.img
```

Then visit the reported localhost:8888 with a browser.

In the singularity command above, the –contain option disables mounting your home directory whereas the -B option specifies paths that will be mounted inside the container, in this case the examples distributed with OMUSE. If you have a folder with notebooks using OMUSE, you can mount this instead of the examples directory to execute them with the singularity container. The container above can also be used to run a regular python script that uses OMUSE functionality by piping the contents to the container python command:

```
cat myscript.py | singularity exec --contain omuse.img python3
```

Finally, it is also possible to launch a shell inside the container:

```
singularity shell --contain -B run:/run/user omuse.img
```

to execute your python code with all OMUSE dependencies findable. We advise to use the –contain option whenever you have OMUSE installed on your host system in $HOME/.local .

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index