SoftwareX 12 (2020) 100608

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

A Python interface to the Dutch Atmospheric Large-Eddy Simulation

Gijs van den Oord [a,*], Fredrik Jansson [b], Inti Pelupessy [a], Maria Chertova [a], Johanna H. Grönqvist [c], Pier Siebesma [d,e], Daan Crommelin [b,f]

[a] Netherlands eScience Center, Science Park 140, 1098XG Amsterdam, The Netherlands
[b] Centrum Wiskunde & Informatica, Science Park 123, 1098XG Amsterdam, The Netherlands
[c] Physics, Faculty of Science and Engineering, Åbo Akademi University, Porthansgatan 3, 20500 Turku, Finland
[d] Center for Civil Engineering and Geosciences, Delft University of Applied Sciences, Stevinweg 1, 2628CN Delft, The Netherlands
[e] Koninklijk Nederlands Meteorologisch Instituut, Utrechtseweg 297, 3731 GA De Bilt, The Netherlands
[f] Korteweg–de Vries Institute for Mathematics, University of Amsterdam, Science Park 105–107, 1098XG Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 13 September 2019
Received in revised form 31 August 2020
Accepted 6 October 2020

Keywords:
Large-eddy simulation
Atmospheric sciences

ABSTRACT

We present a Python interface for the Dutch Atmospheric Large Eddy Simulation (DALES), an existing Fortran code for high-resolution, turbulence-resolving simulation of atmospheric physics. The interface is based on an infrastructure for remote and parallel function calls and makes it possible to use and control the DALES weather simulations from a Python context. The interface is designed within the OMUSE framework, and allows the user to set up and control the simulation, apply perturbations and forcings, collect and analyse data in real time without exposing the user to the details of setting up and running the parallel Fortran DALES code. Another significant possibility is coupling the DALES simulation to other models, for example larger scale numerical weather prediction (NWP) models that can supply realistic lateral boundary conditions. Finally, the Python interface to DALES can serve as an educational tool for exploring weather dynamics, which we demonstrate with an example Jupyter notebook.

Code metadata

Code metadata description	Please fill in this column
Current code version	1.1
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2019_279
Legal Code License	Apache v2.0
Code versioning system used	git
Software code languages, tools, and services used	Fortran 90, Python 3, Singularity, NetCDF4, NumPy, mpi4py, AMUSE, OMUSE, f90nml
Compilation requirements, operating environments & dependencies	Linux, MPI, gcc-gfortran, make, cmake, python3-wheel
If available Link to developer documentation/manual	https://omuse.readthedocs.io/en/latest/
Support email for questions	g.vandenoord@esciencecenter.nl

1. Motivation and significance

Since the advent of numerical weather prediction, many computational models have emerged within the realm of atmospheric sciences. This has resulted in a broad landscape of models, each of them based on approximations and assumptions that are tailored to a typical resolved scale to keep the computational cost within limits. Where general circulation models reproduce large-scale dynamics within resolutions of 10 to 100 km, a large-eddy simulation (LES) is aimed at resolving convective cloud processes and turbulence in the atmosphere, for which a resolution of the order of tens of metres is required; these models therefore typically also assume a limited area domain and vertical extent. The interaction of the small-scale LES with the large scale dynamics has to be provided from an external source, often by specifying forcing

* Corresponding author.
E-mail address: g.vandenoord@esciencecenter.nl (G.v.d. Oord).

https://doi.org/10.1016/j.softx.2020.100608

profiles for the prognostic variables and boundary conditions at the surface. In practice, these parameters have to be present in files that are being read during the simulation.

Our Python interface to the Dutch Atmospheric Large Eddy Simulation (DALES) [1] enables applying these external forcings and boundary conditions in a programmatic way, so that the model can be manipulated during its time stepping. Together with the interface for retrieving the state of the model, this makes it possible to couple DALES to an external agent. One such proven use case [2], and our initial reason for constructing the Python interface to DALES, is the so-called *superparameterization* [3] of the global model OpenIFS [4]. In this scheme, multiple high-resolution DALES instances are coupled to grid columns of OpenIFS, and are used to explicitly simulate cloud and convection processes which are otherwise parametrized in the global model.

However, the applications we envision for the interface layer are much broader than this, since the Python interface to DALES is potentially useful in any application that aims to either (i) drive one or more DALES models with time-dependent forcings where one has full control over the time interpolation without the need to write long and tedious input text files for DALES, (ii) couple DALES instances to other models (with Python interfaces) or (iii) extract specialized diagnostics from DALES, without time-consuming post-processing or modifying the source code.

Finally, we point out that our Python interface to DALES provides an interactive experience which is valuable for educational and exploratory uses of DALES for weather simulations. The software, although being an MPI-parallel Fortran code, can be run from within a user-friendly Python notebook environment thanks to the underlying OMUSE framework [5–7] which provides communication between the Python interface and the computational DALES code. The Python-wrapped DALES model is thus exposed as a stateful, single-threaded Python object and access to its state is seamless despite the distribution of the state over multiple processors. We do stress however that our software does not expose the physical processes and partial tendencies of DALES as separate Python 'building blocks' such as one finds in [8, 9]; rather we provide a lightweight wrapper around the entire model, which perhaps in a future effort may be decomposed at the process level.

2. Software description

2.1. The DALES model

DALES simulates the atmosphere on scales fine enough to resolve cloud and turbulence processes. It does so by numerically solving the conservation laws of momentum, mass, heat and humidity on a rectilinear three-dimensional grid assuming periodic boundary conditions along the horizontal axes, and uses a Fast Fourier Transform to solve the air pressure fluctuations from the Poisson equation. DALES uses second or higher order central difference schemes to model advection and models the subgrid-scale stresses and residuals with eddy viscosities, which are computed either from the turbulence kinetic energy or with a Smagorinsky closure (see e.g. [10]). DALES accounts for all relevant physical processes needed for realistic simulations of cloudy atmospheric conditions, such as thermodynamics, microphysics, radiation and surface–atmosphere interactions.

The program applies an adaptive third-order Runge–Kutta scheme for time stepping. The code is parallelized using the message passing interface (MPI) where the domain is partitioned in either vertical slabs or rectangular columns. DALES also can be forced externally by nudging its mean state towards profiles obtained from observations or another large-scale model.

The Fortran code of DALES is structured in a straightforward and comprehensive way, where all fields are stored globally in a dedicated module and the top-level time stepping loop consists of a sequence of physics routines modelling the processes described above. This makes the code suitable to expose as a simple library with initialization, time stepping, and data access routines.

2.2. Software architecture

Our Python interface to DALES is built using the Python framework OMUSE. It represents DALES with a Python class named `Dales`, enabling interaction with a user or with other Python wrapped models. The interface and the structure of its connections is illustrated in Fig. 1, with the highest-level class `Dales` shown in pink.

OMUSE enables remote procedure calls in Python to programs written in Fortran or C (or any other language with MPI or sockets bindings). The OMUSE framework also provides a number of services to make the deployment of the code as convenient as possible, such as automatic unit conversions, encapsulation of models in object-oriented data objects, an internal state model for wrapped components and proper error handling. These features are all implemented in the Python layer between user code and the model program, and it is up to the developer of a wrapped component to properly configure his Python class to use such services.[1] The OMUSE package contains a collection of predefined Python interfaces to various oceanographic and atmospheric models, giving them consistent interfaces which enables coupling them together or comparing them with each other. The software we present in this paper adds atmospheric modelling to the repertoire of OMUSE, and is now part of the official OMUSE distribution.

The Python definitions of the remote DALES functions are gathered in a Python class named `DalesInterface`. Together with the higher-level functions in the class `Dales`, these form our Python interface to DALES. The interface functions in the class `DalesInterface` each have a Fortran counterpart in the module `dales_interface`. These functions call the DALES original source code routines to handle initialization, getting and setting variable values, and time stepping.

Also the DALES code itself required an additional set of routines in order to be interfaced from Python. The original DALES model was written as a stand-alone program, which performs a simulation according to settings read from a configuration file. To instead control DALES programmatically, we added the possibility to address DALES as a *library*, with functions for initialization, time stepping, retrieving prognostic fields, applying external forcings etc. This functionality is gathered in the new Fortran module `daleslib.f90`, which is included within the DALES source code package. This library version of DALES can also be used independently of OMUSE or Python interfaces, since its functions can be called directly from Fortran. The second modification that has been made is the option to pass an MPI communicator handle to the DALES MPI initialization routine; this is necessary for the integration in OMUSE where the `MPI_COMM_WORLD` is reserved for communication with the master script and models internally use sub-communicators.

When compiling, the DALES source code, the Fortran part of the OMUSE interface and communication functions generated by the OMUSE framework are combined to form a binary called `dales_worker`. When a new `Dales` object is created in Python, OMUSE launches the requested number of `dales_worker` processes by making use of `MPI_COMM_SPAWN`. The worker processes consist of an event loop polling for instructions from the user

[1] For example, by assigning the correct units of DALES data in the OMUSE wrapper, we allow the framework to automatically convert fields to units requested by the user code.

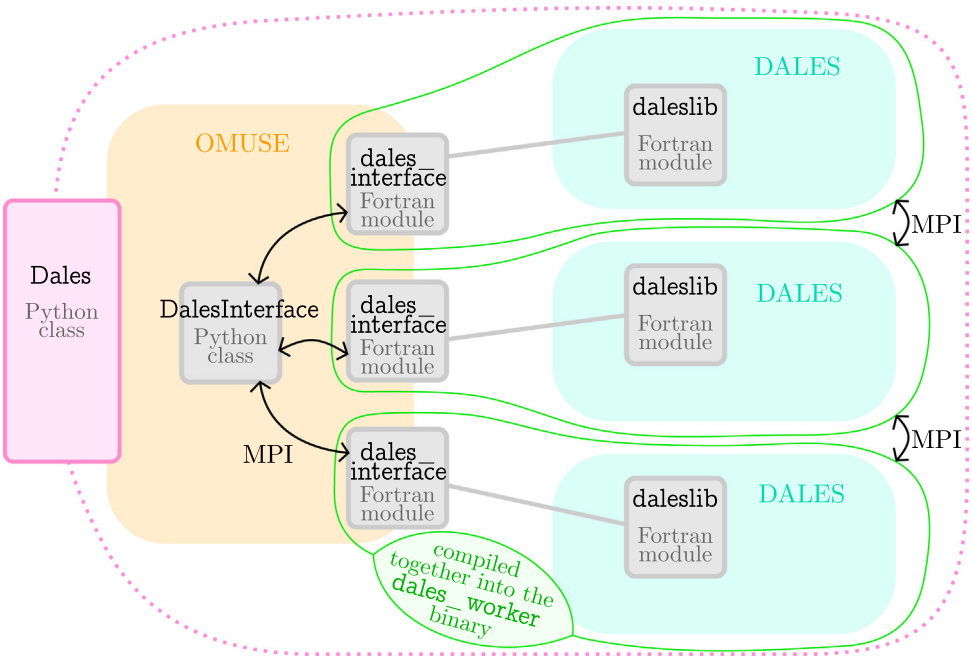

Fig. 1. Overview of the DALES Python interface. The classes `Dales` and `DalesInterface` define the Python interface. Through OMUSE, these call the Fortran functions in `dales_interface`. The `dales_interface` module and the DALES source code are compiled together into a binary called `dales_worker`, denoted by the green lines. Multiple `dales_worker` processes can be launched for a parallel simulation, where each process itself can be (MPI and/or OpenMP) parallel. Here three are shown.

code. The function calls on the `Dales` object in Python are serialized by OMUSE over MPI, and mapped to Fortran routine calls in the remote worker processes. These function calls are used to get and set variable values and to time step the model.

The advantage of this remote procedure design over e.g. a library wrapper with e.g. cython [11] or f2py [12] is that it hides the MPI-parallel nature of `Dales` from the driver script. The philosophy of the interface is to allow users to treat `Dales` as a black box in Python rather than expose all complexity of the code such as parallelization and individual physical processes.

As a consequence of how OMUSE is structured, the Python process does not operate in the same memory space as DALES. This feature has the advantage that multiple independent instances of DALES can be run simultaneously, even though the DALES internal state is stored as a set of global arrays. Furthermore, model instances or model subdomains can run on a different cluster nodes in an HPC environment, communicating over MPI. An obvious drawback is that all data requested through the Python interface will pass through the communication channel, impacting performance if the full 3D grid of data is frequently requested.

In many cases, for example in the superparameterization setup mentioned above, the model coupling is formulated in terms of horizontal averages. For this purpose, the interface provides dedicated functions to request horizontally averaged quantities, resulting in reduced communication volumes compared to averaging the fields on the Python side.

Another performance optimization is provided by the OMUSE framework in the form of non-blocking (asynchronous) versions of the function calls, including the data transfer methods. These can be used to circumvent the Python global interpreter lock and for example to let several model instances time step concurrently (see Appendix B) or exchange data with one model instance while another is performing computations. This feature is essential to obtain a good performance in algorithms running e.g. ensembles of expensive models or to mitigate the costs of data transfers to the master script in multi-model setups.

2.3. Software functionalities

Running a DALES atmospheric simulation using our Python interface involves setting up the model, evolving it over time, and reading or writing the current state of the simulation.

After creating the top-level `Dales` Python object, the user can set model resolution, physical time-independent parameters and initial profiles as attributes to the `Dales` object. The names and the grouping of the time-independent model parameters follows the structure of the DALES configuration Fortran namelist [13].

The input and output variables in the `Dales` Python object, listed in Table A.2, are organized in *grids*, grouping them according to their role in the model and number of dimensions (see Table A.1).

The `Dales` Python object guides the user to call its methods in a sequence that makes physical sense. For example, it is necessary to define the vertical discretization before any vertical forcing profiles can be imposed, and it is also forbidden to change static properties such as the advection scheme after the model has started time-stepping.

To minimize installation effort, we have created a Singularity [14] recipe for a CentOS-based container with DALES, OMUSE and Jupyter [15].

3. Example: DALES simulation of a warm air bubble

As an example of using the Python interface to DALES, we show how to set up and run a simple bubble experiment. In the experiment, the development of a bubble of warm air is studied over time. The resulting image sequence is shown in Fig. 2, where the warmer air is initialized as a sphere near the ground, and then rises upwards with a mushroom-cloud-like appearance.

```
import numpy
import matplotlib.pyplot as plt
from omuse.community.dales.interface import Dales
from omuse.units import units
```

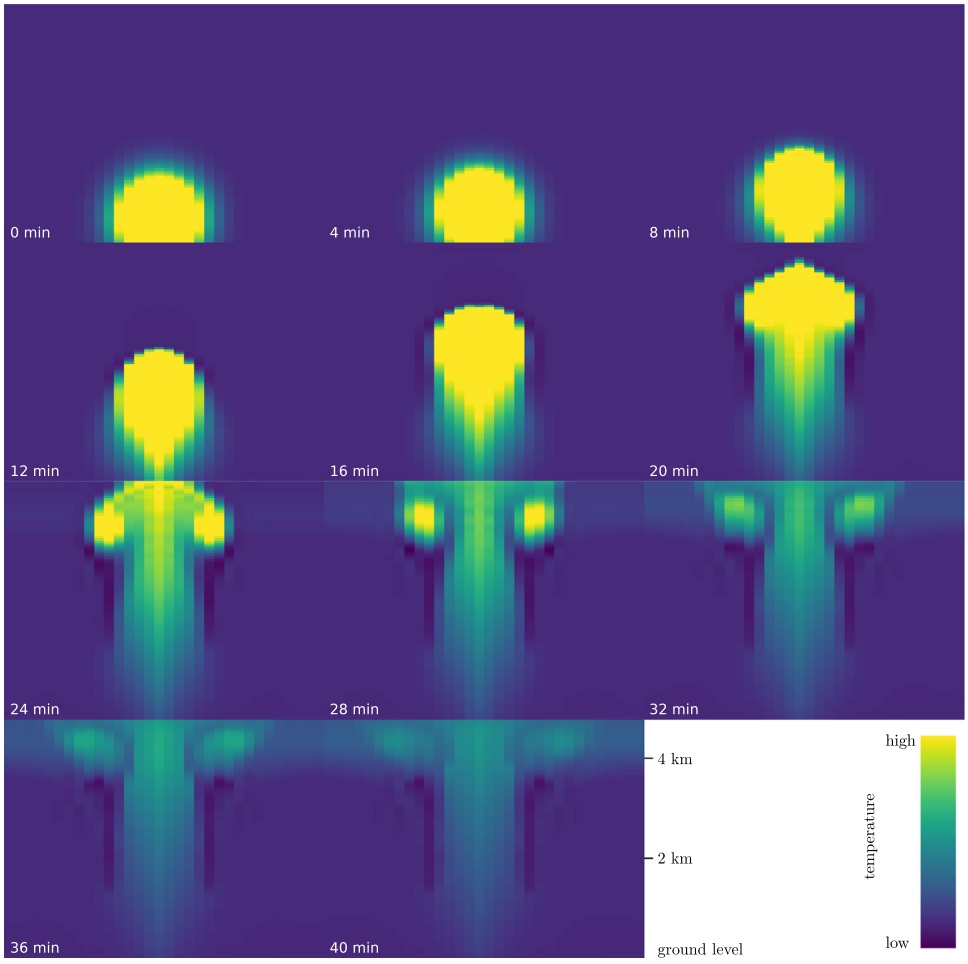

Fig. 2. Warm bubble experiment: vertical cross sections of the air temperature. The initial perturbation is a spherically symmetric shape at ground level. The time series shows the warm air rising, and forming vortices familiar from mushroom clouds as the rise is faster in the middle of the column. This simulation, which takes less than a minute, is performed with the Python script shown in the text. The temperature shown is the liquid water potential temperature — which is the temperature quantity DALES uses internally.

```
# create Dales object
d=Dales(workdir='daleswork', channel_type='sockets',
  number_of_workers=1)
# add redirection='none' to see the model log messages

# Set parameters: domain size and resolution, advection
  scheme
d.parameters_DOMAIN.itot = 32  # number of grid cells in x
d.parameters_DOMAIN.jtot = 32  # number of grid cells in y
d.parameters_DOMAIN.xsize = 6400 | units.m
d.parameters_DOMAIN.ysize = 6400 | units.m
d.parameters_DYNAMICS.iadv_mom = 6 # 6th order advection
  for momentum
d.parameters_DYNAMICS.iadv_thl = 5 # 5th order advection
  for temperature
d.parameters_RUN.krand = 0  # initial state randomization
  off

d.parameters_RUN.ladaptive = True
d.parameters_RUN.courant  = 0.5
d.parameters_RUN.peclet   = 0.1

d.parameters_PHYSICS.lcoriol = False
d.parameters_PHYSICS.igrw_damp = 3

# initialize all velocities to 0 and a low spec. humidity
d.fields[:,:,:].U = 0 | units.m / units.s
d.fields[:,:,:].V = 0 | units.m / units.s
d.fields[:,:,:].W = 0 | units.m / units.s
d.fields[:,:,:].QT = 0.001 | units.kg / units.kg

# add perturbation in temperature - Gaussian bubble at
  (cx,cy,cz), radius r
cx,cy,cz,r = 3200|units.m, 3200|units.m, 500|units.m,
             500|units.m
d.fields[:,:,:].THL += (0.5 | units.K) * numpy.exp(
    -((d.fields.x-cx)**2 + (d.fields.y-cy)**2
    + (d.fields.z-cz)**2)/(2*r**2))

times = numpy.linspace(0, 44, 12) | units.minute # times
        for snapshots
fig, axes = plt.subplots(3, 4, sharex=True, sharey=True)
extent = (0, d.fields.y[0,-1,0].value_in(units.m),
          0, d.fields.z[0,0,-1].value_in(units.m))
for t,ax in zip(times, axes.flatten()):
    print('Evolving to', t)
    d.evolve_model(t)
    thl = d.fields[:,:,:].THL
    wthl = d.fields[:,:,:].W * thl
    kwtmax = numpy.unravel_index(numpy.argmax(numpy.
             abs(wthl)), wthl.shape)[2]
```

```
    zwtmax = d.profiles.z[kwtmax]
    print("Height of the maximal heat flux is at", zwtmax)
    im = ax.imshow(thl[16,:,:].value_in(units.K).
         transpose(), extent=extent,
    origin='lower', vmin=292.5, vmax=292.75)
    ax.text(.1, .1, str(t.in_(units.minute)),
            color='w', transform=ax.transAxes)
plt.show()
```

4. Impact

As the Python language has become the dominant scripting language in scientific computing and data analysis, running experiments and accessing the model state from within Python will prove to be a valuable asset to users of high-resolution weather models, in the present case, users of the DALES software specifically. Our Python interface supports procedures like setting up a high-resolution weather simulation, as well as nudging it in real time towards observed atmospheric profiles.

Usually these profiles originate from observations or large-scale weather model output, and using the Python interface saves the user from the tedious job of writing the DALES input files in the appropriate format. In this sense, the Python interface enables experimentation and rapid prototyping with the model.

The Python interface also provides a front-end to DALES that is suitable for educational purposes. The possibility to manipulate DALES interactively within a Jupyter notebook helps students gain insight in topics like the thermodynamics of clouds, atmospheric convection, surface processes and boundary layer turbulence.

The most significant added value of a library interface, however, is in coupling with other models. By encapsulating DALES in the OMUSE framework, there is a clear path to integration with other environmental software. One example of this is the superparameterization of the global weather and climate model OpenIFS, mentioned in Section 1, where multiple high-resolution DALES instances are coupled to grid columns of the global model.

The advantage of the coupling strategy of OMUSE versus more implicit and less intrusive approaches like OASIS [16] is the expressive nature of the control script setup. The equations governing the coupling and time integration scheme can be easily read and modified in the Python code because the objects contain recognizable methods, and the data transfers occur via NumPy [17] arrays with familiar names, as opposed to more generic frameworks like the model coupling toolkit of Ref. [18].

As the interface enables one to extract tailored diagnostics from DALES, it may be used to offer high-resolution atmospheric boundary conditions to other environmental models. For example, the precipitation fluxes in DALES can be coupled to fine-scale hydrological models for flood risk assessment in future climate scenarios. The DALES surface fields and fluxes can also be coupled to advanced surface dynamics models to study realistic surface–cloud feedback processes, and the momentum fluxes can be coupled to wind stresses in coastal hydrodynamics models. Furthermore, the passive tracers in DALES can be coupled to external atmospheric chemistry or air quality models, without the need to integrate them into the DALES Fortran source code.

Finally, the Python interface to DALES opens up the possibility to integrate DALES into other complex workflows, such as downscaling external forcings and extracting dedicated diagnostics as needed in the forecasting of renewable energy yields, or the training of machine learning algorithms onto DALES output to construct fast surrogate models.

5. Conclusions

We have constructed Fortran and Python interfaces to the DALES program for interactive high-resolution weather modelling. The interface allows the user to retrieve data from DALES and manipulate the model dynamically from a scripting front-end. This functionality increases the usability of DALES significantly, and allows the code to be coupled to other earth system models. One such proven use case is the superparameterization of the global weather model OpenIFS, where multiple DALES instances are coupled to grid columns of the global weather model. Furthermore, the interface facilitates the use of the model for educational purposes, or in more complex workflows. The interface is object-oriented, contains familiar methods to access the model state, and allows creating multiple DALES instances, with full control over the occupation of the available hardware resources.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was supported by the Netherlands eScience Center (NLeSC) under grant no. 027.015.G03.

Appendix A. Table of model variables

See Tables A.1 and A.2.

Appendix B. Asynchronous requests to DALES example

In this section we illustrate the asynchronous requests functionality with a very basic example time stepping two DALES instances concurrently. To establish this, one should create a requests pool and call the 'asynchronous' versions of `evolve_model` method.

```
from amuse.rfi.async_request import AsyncRequestPool
# In this code, we assume two instances of the Dales
  Python class,
# dales1 and dales2, have been created and initialized
pool = AsyncRequestsPool()
nexttime = dales1.get_model_time() + 300 | units.s
req1 = dales1.evolve_model.asynchronous(nexttime)
pool.add_request(req1)
req2 = dales2.evolve_model.asynchronous(nexttime)
pool.add_request(req2)
req3 = dales2.get_profile_THL.asynchronous()
pool.add_request(req3)
pool.waitall() # Wait until all asynchronous calls are
 finished
thlprof = req3.result()
```

In the code above, the θ_ℓ profile retrieval is executed asynchronously w.r.t. the master script too, but the pool ensures it is issued only after the evolve of `dales2` has been finished.

Table A.1
Organization of data grids in the DALES Python API. The operator $\langle \dots \rangle_{xy}$ denotes horizontal averaging of volume fields.

Grid name	Description	Read/write	Variables
`fields`	3D prognostic variables	w	u, v, w, θ_ℓ, q_t
`fields`	3D general variables	r	u, v, w, θ_ℓ, q_t, q_ℓ, q_i, q_r, q_{sat}, $\sqrt{e}$, T, π, $F^{\uparrow,\downarrow}_{S,L}$, $C^{\uparrow,\downarrow}_{S,L}$, F_{dir}, F_{dif}
`profiles`	Horizontally averaged fields	r	$\langle u \rangle_{xy}$, $\langle v \rangle_{xy}$, $\langle w \rangle_{xy}$, $\langle \theta_\ell \rangle_{xy}$, $\langle q_t \rangle_{xy}$, $\langle q_\ell \rangle_{xy}$, $\langle q_r \rangle_{xy}$, $\langle \sqrt{e} \rangle_{xy}$, $\langle T \rangle_{xy}$, p, ρ, A
`forcing_profiles`	Forcing profiles	w	$\langle u \rangle_{xy}$, $\langle v \rangle_{xy}$, $\langle \theta_\ell \rangle_{xy}$, $\langle q_t \rangle_{xy}$
`nudging_profiles`	Nudging profiles	w	$\langle u \rangle_{xy}$, $\langle v \rangle_{xy}$, $\langle \theta_\ell \rangle_{xy}$, $\langle q_t \rangle_{xy}$
`scalars`	Uniform fields	rw	p_s, $\langle z_m \rangle_{xy}$, $\langle z_h \rangle_{xy}$, $\langle \overline{w\,\theta} \rangle_{xy}$, $\langle \overline{w\,q} \rangle_{xy}$
`surface_fields`	Horizontal fields	r	lwp, twp, rwp, u_*, z_m, z_h, T_{skin}, q_{skin}, Q_s, Q_l, Λ, $\overline{w\,q_t}$, $\overline{w\,\theta_\ell}$

Table A.2
List of DALES variables exposed in the Python wrapper.

Symbol	Unit	Dimensions	Attribute	Variable description
u, v, w	m/s	*xyz*	`U`, `V`, `W`	East-, north- and upward air velocity
θ_ℓ	K	*xyz*	`THL`	Liquid water potential temperature
q_t	kg/kg	*xyz*	`QT`	Total specific humidity
$\sqrt{e}$	m/s	*xyz*	`E12`	Turbulence kinetic energy
T	K	*xyz*	`T`	Air temperature
q_ℓ, q_i, q_r	kg/kg	*xyz*	`QL`, `QL_ice`, `QR`	Liquid, ice and rain water content
lwp, twp, rwp	kg/m^2	*xy*	`LWP`, `TWP`, `RWP`	Liquid, total and rain water paths
q_{sat}	kg/kg	*xyz*	`Qsat`	Saturation humidity
π	m^2/s^2	*xyz*	`pi`	Modified air pressure
ρ	kg/m^2	*z*	`rho`	Air density
p	Pa	*z*	`P`	Hydrostatic air pressure
A	m^2/m^2	*z*	`A`	Cloud fraction profile
$F^{\uparrow,\downarrow}_{S,L}$	W/m^2	*xyz*	`r{s,l}w{u,d}`	Up- and downwelling short- and longwave radiative fluxes
$C^{\uparrow,\downarrow}_{S,L}$	W/m^2	*xyz*	`r{s,l}w{u,d}cs`	Clear-sky up- and downwelling short- and longwave radiative fluxes
F_{dir}, F_{dif}	W/m^2	*xyz*	`rswdir`, `rswdif`	Downwelling shortwave direct and diffuse radiative fluxes
T_{skin}	K	*xy*	`tskin`	Skin temperature
q_{skin}	kg/kg	*xy*	`qskin`	Skin humidity
$\overline{w\theta_\ell}$	mK/s	*xy*	`wt`	Surface θ_ℓ flux
$\overline{wq_t}$	m/s	*xy*	`wq`	Surface specific humidity flux
Q_s, Q_l	W/m^2	*xy*	`H`, `LE`	Sensible and latent heat fluxes
Λ	m	*xy*	`obl`	Obukhov length
u_*	m/s	*xy*	`ustar`	Friction velocity
z_m, z_h	m	*xy*	`z0m`, `z0h`	Roughness lengths for momentum and heat

References

[1] Heus T, van Heerwaarden CC, Jonker HJJ, Pier Siebesma A, Axelsen S, van den Dries K, Geoffroy O, Moene AF, Pino D, de Roode SR, Vilà-Guerau de Arellano J. Formulation of the dutch atmospheric large-eddy simulation (DALES) and overview of its applications. Geosci Model Dev 2010;3(2):415–44. http://dx.doi.org/10.5194/gmd-3-415-2010.

[2] Jansson F, van den Oord G, Pelupessy I, Grönqvist JH, Siebesma AP, Crommelin D. Regional superparameterization in a global circulation model using large eddy simulations. J Adv Model Earth Syst http://dx.doi.org/10.1029/2018MS001600.

[3] Grabowski WW. Coupling cloud processes with the large-scale dynamics using the cloud-resolving convection parameterization (CRCP). J Atmos Sci 2001;58(9):978–97. http://dx.doi.org/10.1175/1520-0469(2001)058<0978:CCPWTL>2.0.CO;2.

[4] Carver G, et al. The ECMWF OpenIFS numerical weather prediction model release cycle 40r1: description and use cases. 2018, in preparation to be submitted to GMD.

[5] Zwart SFP, McMillan SL, van Elteren A, Pelupessy FI, de Vries N. Multi-physics simulations using a hierarchical interchangeable software interface. Comput Phys Comm 2013;184(3):456–68. http://dx.doi.org/10.1016/j.cpc.2012.09.024.

[6] Pelupessy I, van Werkhoven B, van Elteren A, Viebahn J, Candy A, Portegies Zwart S, Dijkstra H. The oceanographic multipurpose software environment (OMUSE v1.0). Geosci Model Dev 2017;10(8):3167–87. http://dx.doi.org/10.5194/gmd-10-3167-2017.

[7] Pelupessy I, Portegies Zwart S, van Elteren A, Dijkstra H, Jansson F, Crommelin D, Siebesma P, van Werkhoven B, van den Oord G. Creating a reusable cross-disciplinary multi-scale and multi-physics framework: From AMUSE to OMUSE and beyond. In: Rodrigues JMF, Cardoso PJS, Monteiro J, Lam R, Krzhizhanovskaya VV, Lees MH, Dongarra JJ, Sloot PM, editors. Computational Science – ICCS 2019. Cham: Springer International Publishing; 2019, p. 379–92.

[8] Monteiro JM, McGibbon J, Caballero R. Sympl (v. 0.4.0) and climt (v. 0.15.3) – towards a flexible framework for building model hierarchies in Python. Geosci Model Dev 2018;11(9):3781–94. http://dx.doi.org/10.5194/gmd-11-3781-2018.

[9] Rose B. CLIMLAB: a Python toolkit for interactive, process-oriented climate modeling. J Open Sour Softw 2018;3(24):659. http://dx.doi.org/10.21105/joss.00659.

[10] Schmidt H, Schumann U. Coherent structure of the convective boundary layer derived from large-eddy simulations. J Fluid Mech 1989;200:511–62.

[11] Behnel S, Bradshaw R, Citro C, Dalcin L, Seljebotn DS, Smith K. Cython: The best of both worlds. Comput Sci Eng 2011;13(2):31–9.

[12] Peterson P. F2py: a tool for connecting fortran and python programs. Int J Comput Sci Eng 2009;4(4):296–305.

[13] Heus T, van Heerwaarden C, van der Dussen J, Ouwersloot H. Overview of all namoptions in DALES. 2019, Accessed: 2019-07-25 https://github.com/dalesteam/dales/blob/master/utils/doc/input/Namoptions.pdf.

[14] Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. Plos One 2017;12(5):1–20. http://dx.doi.org/10.1371/journal.pone.0177459.

[15] Kluyver T, Ragan-Kelley B, Pérez F, Granger B, Bussonnier M, Frederic J, Kelley K, Hamrick J, Grout J, Corlay S, et al. Jupyter notebooks—a publishing format for reproducible computational workflows. In: Positioning and Power in Academic Publishing: Players, Agents and Agendas: Proceedings of the 20th International Conference on Electronic Publishing. IOS Press; 2016, p. 87.

[16] Valcke S. The OASIS3 coupler: a European climate modelling community software. Geosci Model Dev 2013;6(2):373–88. http://dx.doi.org/10.5194/gmd-6-373-2013.

[17] Dubois PF, Hinsen K, Hugunin J. Numerical python. Comput Phys 1996;10(3):262–7. http://dx.doi.org/10.1063/1.4822400.

[18] Larson J, Jacob R, Ong E. The model coupling toolkit: A new fortran90 toolkit for building multiphysics parallel coupled models. Int J High Perform Comput Appl 2005;19(3):277–92. http://dx.doi.org/10.1177/1094342005056115.