A Python interface to the Dutch Atmospheric Large-Eddy Simulation

van den Oord, G.; Jansson, F.; Pelupessy, I.; Chertova, M.; Grönqvist, J.H.; Siebesma, P.; Crommelin, D.

DOI
10.1016/j.softx.2020.100608

Publication date
2020

Document Version
Final published version

Published in
SoftwareX

License
CC BY

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
A Python interface to the Dutch Atmospheric Large-Eddy Simulation

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

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

\section*{A R T I C L E  I N F O}

Article history:
Received 3 September 2019
Received in revised form 31 August 2020
Accepted 6 October 2020

Keywords:
Large-eddy simulation
Atmospheric sciences

\section*{A B S T R A C T}

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.

© 2020 Netherlands eScience Center. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

\section*{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

...
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 parameterized 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. 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_WORLD. 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.
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.

```python
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.

```python
# 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.ldapaptive = 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)
estent = (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
kwmax = numpy.unravel_index(numpy.argmax(numpy.abs(wthl)), wthl.shape)[2]
```

4
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 frontend. 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.

```python
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 = AsyncRequestPool()
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 $\theta_l$ 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 \((\ldots)_x\) denotes horizontal averaging of volume fields.

<table>
<thead>
<tr>
<th>Grid name</th>
<th>Description</th>
<th>Read/write</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields</td>
<td>3D prognostic variables</td>
<td>w</td>
<td>( u, v, w, \theta, q, )</td>
</tr>
<tr>
<td>profiles</td>
<td>3D general variables</td>
<td>r</td>
<td>( u, v, w, \theta, q, \bar{q}, q_v, q_a, \sqrt{T}, T, \tau, F_v, C_v, F_s, \tilde{F}_s )</td>
</tr>
<tr>
<td>forcing_profiles</td>
<td>Horizontally averaged fields</td>
<td>r</td>
<td>( u, v, w, (\theta)_y, (\bar{q})_y, (q_v)_y, (q_a)_y, \sqrt{T}_y, (T)_y, B, \rho, A )</td>
</tr>
<tr>
<td>nudging_profiles</td>
<td>Nudging profiles</td>
<td>w</td>
<td>( u, v, w, (\theta)_y, (\bar{q})_y, (q_v)_y )</td>
</tr>
<tr>
<td>scalars</td>
<td>Uniform fields</td>
<td>rw</td>
<td>( p, (z_{\text{prof}})<em>y, (\theta)</em>{\text{prof}} )</td>
</tr>
<tr>
<td>surface_fields</td>
<td>Horizontal fields</td>
<td>r</td>
<td>( lwp, twp, rwp, u_s, z_h, z_0, F_{\text{dir}}, Q_{\text{dir}}, Q_v, Q_a, A, \sqrt{\Omega}, \sqrt{\gamma}, )</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Unit</th>
<th>Dimensions</th>
<th>Attribute description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u, v, w )</td>
<td>m/s</td>
<td>xy</td>
<td>East-, north- and upward air velocity</td>
</tr>
<tr>
<td>( \theta )</td>
<td>K</td>
<td>xyz</td>
<td>Liquid water potential temperature</td>
</tr>
<tr>
<td>( q_r, q_v, q_a)</td>
<td>kg/kg</td>
<td>xy</td>
<td>Total specific humidity</td>
</tr>
<tr>
<td>( T )</td>
<td>m/s</td>
<td>xy</td>
<td>Turbulence kinetic energy</td>
</tr>
<tr>
<td>( q )</td>
<td>K</td>
<td>xyz</td>
<td>Air temperature</td>
</tr>
<tr>
<td>( \pi )</td>
<td>m/s²/s³</td>
<td>xy</td>
<td>Liquid, ice and rain water content</td>
</tr>
<tr>
<td>( \rho )</td>
<td>m/kg³</td>
<td>z</td>
<td>Liquid, total and rain water paths</td>
</tr>
<tr>
<td>( \rho_{\text{a}} )</td>
<td>kg/kg</td>
<td>xy</td>
<td>Modified air pressure</td>
</tr>
<tr>
<td>( \rho_{\text{d}} )</td>
<td>Kg/kg</td>
<td>xy</td>
<td>Modified air pressure</td>
</tr>
<tr>
<td>( A )</td>
<td>m²/m²</td>
<td>z</td>
<td>Cloud fraction profile</td>
</tr>
<tr>
<td>( F_v, C_v )</td>
<td>W/m²</td>
<td>xy</td>
<td>Up- and downwelling short- and longwave radiative flux</td>
</tr>
<tr>
<td>( F_{\text{dir}}, F_s )</td>
<td>W/m²</td>
<td>xy</td>
<td>Downwelling shortwave direct and diffuse radiative flux</td>
</tr>
<tr>
<td>( T_{\text{skin}} )</td>
<td>K</td>
<td>xy</td>
<td>Skin temperature</td>
</tr>
<tr>
<td>( q_{\text{skin}} )</td>
<td>Kg/kg</td>
<td>xy</td>
<td>Skin humidity</td>
</tr>
<tr>
<td>( \eta_{\text{rms}} )</td>
<td>K</td>
<td>xy</td>
<td>Surface ( \theta ) flux</td>
</tr>
<tr>
<td>( w_{\text{rms}} )</td>
<td>m/s</td>
<td>xy</td>
<td>Surface specific humidity flux</td>
</tr>
<tr>
<td>( Q_v, Q_a )</td>
<td>W/m²</td>
<td>xy</td>
<td>Sensible and latent heat flux</td>
</tr>
<tr>
<td>( A )</td>
<td>m</td>
<td>xy</td>
<td>Obukhov length</td>
</tr>
<tr>
<td>( u_s )</td>
<td>m/s</td>
<td>xy</td>
<td>Friction velocity</td>
</tr>
<tr>
<td>( z_0, z_h )</td>
<td>m</td>
<td>xy</td>
<td>Roughness lengths for momentum and heat</td>
</tr>
</tbody>
</table>

References


