fv3gfs-wrapper: a Python wrapper of the FV3GFS atmospheric model

Jeremy McGibbon[1], Noah D. Brenowitz[1], Mark Cheeseman[1], Spencer K. Clark[1,2], Johann Dahm[1], Eddie Davis[1], Oliver D. Elbert[1,2], Rhea C. George[1], Lucas M. Harris[2], Brian Henn[1], Anna Kwa[1], W. Andre Perkins[1], Oliver Watt-Meyer[1], Tobias Wicky[1], Christopher S. Bretherton[1,3], and Oliver Fuhrer[1]

[1]Vulcan Inc., Seattle, WA
[2]Geophysical Fluid Dynamics Laboratory, NOAA, Princeton, NJ
[3]Department of Atmospheric Sciences, University of Washington, Seattle, WA

Correspondence: Jeremy McGibbon (mcgibbon@uw.edu)

Abstract. Simulation software in geophysics is traditionally written in Fortran or C++ due to the stringent performance requirements these codes have to satisfy. As a result, researchers who use high-productivity languages for exploratory work often find these codes hard to understand, hard to modify, and hard to integrate with their analysis tools. `fv3gfs-wrapper` is an open-source Python-wrapped version of the NOAA (National Oceanic and Atmospheric Administration) FV3GFS (Finite-Volume Cubed Sphere Global Forecasting System) global atmospheric model, which is coded in Fortran. The wrapper provides simple interfaces to progress the Fortran main loop and get or set variables used by the Fortran model. These interfaces enable a wide range of use cases such as modifying the behavior of the model, introducing online analysis code, or saving model variables and reading forcings directly to and from cloud storage. Model performance is identical to the fully-compiled Fortran model, unless routines to copy state in and out of the model are used. This copy overhead is well within an acceptable range of performance, and could be avoided with modifications to the Fortran source code. The wrapping approach is outlined and can be applied similarly in other Fortran models to enable more productive scientific workflows.

1 Introduction

FV3GFS (Finite-Volume Cubed Sphere Global Forecasting System) (Zhou et al., 2019) is a prototype of the operational Global Forecast System of the National Centers for Environmental Prediction. In this document when we say FV3GFS we are referring specifically to the atmospheric component of the U. S. National Oceanic and Atmospheric Administration (NOAA) Unified Forecast System (UFS, https://ufscommunity.org/) for operational numerical weather prediction. We forked this code from the v1 branch of the UFS model in December 2019. It uses the Geophysical Fluid Dynamics Laboratory (GFDL) Finite-Volume Cubed-Sphere Dynamical Core (FV3). FV3 solves the non-hydrostatic equations of atmospheric motion discretized on a cubed sphere using a finite volume scheme on a terrain-following grid with D-grid wind staggering (Putman and Lin, 2007; Harris and Lin, 2013). The model is written in Fortran (Global Engineering Documents, 1991) and parallelized using a hybrid OpenMP (Open Multi-Processing, OpenMP Architecture Review Board (2020)) / MPI (Message Passing Interface, Message Passing Interface Forum (2015)) approach, which allows for performant execution through compilation.

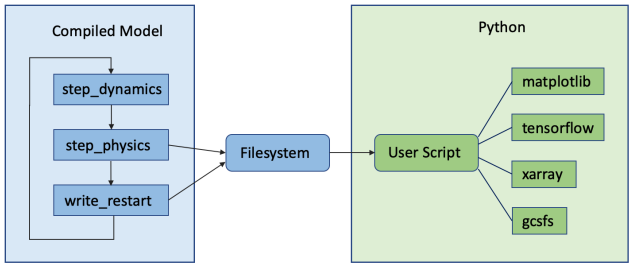

Figure 1. Schematic of Fortran-centric workflow using the filesystem to transfer data to Python user code. Arrowheads indicate the direction of the model main loop, as well as data transfer out of the Fortran model and into the Python user script.

However, development of an atmospheric model using a low-level, strongly typed programming language with a small user base has trade-offs. Libraries for interacting with cloud storage, performing physical or statistical analysis, and using machine learning are not as readily available or widely used in languages like Fortran as they are in high-level languages such as Python. A Python interface to the compiled Fortran code can enable a much larger user base to interact with this code, and allow a large ecosystem of Python tools to be interfaced with model routines.

Python is often integrated into Fortran modeling workflows as a post-processing tool, as shown in Figure 1. In this workflow, Python is used to perform computations on data saved to the filesystem by the Fortran model. This approach has several shortcomings. It is rarely feasible to store the full-resolution model state at each model time step, so often statistics over time are stored instead. Unless sufficiently frequent snapshots are stored, computing new statistics directly from full-resolution instantaneous fields requires writing Fortran code to run in the model. This can be an issue if developer documentation is not available or the user is not familiar with Fortran. This approach requires writing to disk before data can be used in Python, which may be unnecessary if the written data is not a necessary end product. Such filesystem operations can be a significant bottleneck in computation time. This approach also does not provide a way to use Python libraries when modifying the behavior of the Fortran model, as any logic after the data is read from disk must be written in Fortran. Instead, machine learning practitioners port machine learning routines to Fortran using models that have been trained and saved using Python (Ott et al., 2020; Curcic, 2019).

In this work, we present a Python wrapper for the FV3GFS global atmospheric model. As shown in Figure 2, the FV3GFS model is compiled as a shared library with wrapper routines that provide an API to control and interact with the model. At the core of any weather or climate model is the main integration loop, which integrates the model state forward by a period if time. The wrapper splits the simple model main loop into a sequence of subroutines that can be called from Python. This allows the main loop to be written in Python, through calls to each section of the Fortran main loop (`step_dynamics`, `step_physics`). Furthermore, it allows copying variables into (`set_state`) or out of (`get_state`) the Fortran runtime environment, so it can be used in Python functions that can affect the integration of the Fortran model state. Data retrieved with `get_state` includes units information, for ease of debugging and for reference on data written to disk.

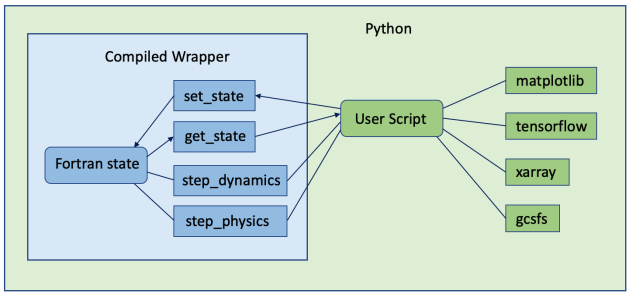

Figure 2. Schematic of Python-centric workflow using fv3gfs-wrapper, showing how it can interface with Python libraries during model execution. Arrowheads indicate data transfer between user Python code and Fortran model.

As the wrapper currently stands, configuration is deferred entirely to the Fortran model code. The only change in initialization is that MPI is initialized by mpi4py, after which the MPI communicator is passed as a Fortran handle to the model initialization routines. This allows us to maintain feature completeness with the existing Fortran model, without re-writing configuration logic.

This Python-centric workflow enables a fundamentally different way to integrate tools available in the Python ecosystem into a Fortran modeling workflow. A user can add online diagnostic code after a physics or dynamics step, and perform input or output (I/O) to or from cloud resources. The model state can be reset to a previous one, allowing sensitivity studies to be run online. Custom logic can be added to the main loop after a physics or dynamics step, such as a machine learning corrector parameterization or nudging to a cloud-based forcing dataset. The use of a Python main loop makes it significantly easier to integrate custom I/O, diagnostic routines, and physical parameterizations into the model.

This ease of integration is an important tool when developing parameterizations using machine learning. When developing such schemes, offline performance does not guarantee performance when run online within a model. However it can be difficult to rapidly train a model in a language with machine learning libraries such as Python and then convert it for use in Fortran. Solutions have so far been based on Fortran executables, either by calling Python from Fortran (Brenowitz and Bretherton, 2019) or by re-implementing neural network codes in Fortran (Ott et al., 2020; Curcic, 2019). Because of the strong tooling available for machine learning in Python, it is advantageous to be able to include Python machine learning code within the atmospheric model. Presently Python code can only be integrated outside of the dynamics and physics routines, and not within the physics suite. Adding flexibility to introduce Python code between individual physics schemes remains a subject for future work.

This is not the first time Python and high-performance compiled programs have been combined. qtcm (Lin, 2009) applies a similar wrapping approach to a quasi-equilibrium tropical circulation model using f2py (Peterson, 2009), an automated Fortran to Python interface generator. PyCLES (Pressel et al., 2015) is a full large-eddy simulation written in Cython, a variant of Python that compiles to C code and can interoperate with Python codes. CliMT (Monteiro et al., 2018) wraps Fortran model components into Python objects that can be composed to define a model main loop in Python. In astronomy, Python

computational codes such as nbodykit (Hand and Feng, 2017) run using numpy (Harris et al., 2020) and MPI for Python (Dalcín et al., 2008), and are shown to scale to thousands of ranks. These previous works provide confidence that a model using Python to call compiled code can provide the level of scaling and performance required for atmospheric and climate science research.

A consideration in designing new atmospheric models is the large amount of legacy Fortran code already available. As a consequence, new model components are often written in Fortran so they can interface with such legacy code. Efforts to re-write existing Fortran models (for example, to run on Graphics Processing Unit (GPU) architectures), can benefit from the ability to progressively replace existing components with refactored or re-written codes in other languages.

To motivate the design choices made in this work, we present our main priorities: retain existing functionality of the Fortran model, minimal sacrifice of performance, a main time stepping loop which is easy to understand and modify, and minimal changes to Fortran code.

Most of these priorities clearly come from our focus on improving model accessibility for researchers interested in modifying the behavior of the Fortran code. They would benefit from retaining the existing functionality they would like to modify, and they should be able to easily understand how the code can be modified. They may require efficient model performance on high-performance computers for research problems using higher-resolution simulations. By minimizing the needed changes to the Fortran code, we can reduce the effort required to switch to a new Fortran model version.

While this wrapper has many applications, we will focus on illustrative scenarios relevant to our own FV3GFS model development work. In addition to reproducing the existing model behavior, we will show how to: augment the Fortran model with a machine learning parameterization, include custom MPI communication as part of online diagnostic code, and perform online analysis in a Jupyter notebook.

We will begin by showing in Section 2 how fv3gfs-wrapper can be used to reproduce, bit-for-bit, the results of the existing Fortran model. We will then show in Section 3 how fv3gfs-wrapper enables each of these use cases while achieving our priorities of performance, ease of understanding, and ease of modification. Having presented the features of fv3gfs-wrapper by example, we will delve more deeply into their implementation in Section 4. Finally, we will discuss some of the challenges encountered in designing and implementing fv3gfs-wrapper in Section 4.5 before drawing our conclusions in Section 5.

2 Validation

For completeness and testing, fv3gfs-wrapper should be able to reproduce, bit-for-bit, the results of the Fortran model. This allows us to test the logic wrapping the Fortran code. Because the wrapper executes Fortran code identical to the original Fortran model, bit-for-bit regression on one parameter configuration or forcing dataset gives us confidence the code can be used for any parameter configuration or forcing dataset. The implementation of this use case is as follows:

```
import fv3gfs.wrapper

if __name__ == "__main__":
    fv3gfs.wrapper.initialize()
    for i in range(fv3gfs.wrapper.get_step_count()):
```

Table 1. Run times of examples and compiled Fortran model. Baseline refers to reproducing existing Fortran behavior. Examples were run for 6 hours of simulation time at C48 resolution on 6 processors on a 2019 Macbook Pro. Each example was run three times, and the shortest time is reported.

Example	Run time (s)
Fortran baseline	110
Wrapper baseline	110
Random Forest	116
Minimum Surface Pressure	110

```
        fv3gfs.wrapper.step_dynamics()
        fv3gfs.wrapper.step_physics()
        fv3gfs.wrapper.save_intermediate_restart_if_enabled()
    fv3gfs.wrapper.cleanup()
```

The existing main routine in `coupler_main.f90` separates relatively cleanly into five routines: one each to initialize and finalize the model, one for dynamics (resolved fluid flow), one for physics (subgrid-scale processes), and one that will write intermediate restart data if intermediate restart files are enabled for the run and if we should write a restart on the current timestep. Each of these Python routines calls that section of the Fortran code, and then returns to a Python context.

The overhead of the Python time step loop and the wrapper functions is negligible in comparison to the computation done within a process (Table 1), meeting our performance goal. The conciseness of the main loop makes it easy to understand what the code is doing at a high level. This example is easy to modify, as shown in the use cases in the next section.

This code and the command-line examples below are available in the examples/gmd_timings directory of the git repository for fv3gfs-wrapper as referenced in the Code and data availability statement, using a 6-hour C48 run directory available as McGibbon et al. (2021d). The timings for each of these examples are included in Table 1. We can see the wrapper does not add significant overhead to the Fortran baseline timing.

3 Use cases in action

All examples discussed in this section are included in the public repository for fv3gfs-wrapper linked in the Code and data availability statement. We encourage the reader to download and run these examples on their own computer, using the example run directory available as McGibbon et al. (2021d).

3.1 Augmenting the model with machine learning

An important use case motivating this work is to be able to modify the operation of the model main loop, for example by adding a machine learning model that applies tendencies at the end of each timestep. This serves as an example for how the main loop

can be modified more generically, such as by adding I/O functionality or online diagnostics, using `fv3gfs.wrapper.get_state` and `fv3gfs.wrapper.set_state` to interface with the Fortran model.

```
import fv3gfs.wrapper
import fv3gfs.wrapper.examples
from datetime import timedelta
import f90nml

if __name__ == "__main__":
    # load timestep from the namelist
    namelist = f90nml.read("input.nml")
    timestep = timedelta(seconds=namelist["coupler_nml"]["dt_atmos"])
    # initialize the machine learning model
    rf_model = fv3gfs.wrapper.examples.get_random_forest()
    fv3gfs.wrapper.initialize()
    for i in range(fv3gfs.wrapper.get_step_count()):
        fv3gfs.wrapper.step_dynamics()
        fv3gfs.wrapper.step_physics()

        # apply an update from the machine learning model
        state = fv3gfs.wrapper.get_state(rf_model.inputs)
        rf_model.update(state, timestep=timestep)
        fv3gfs.wrapper.set_state(state)

        fv3gfs.wrapper.save_intermediate_restart_if_enabled()
    fv3gfs.wrapper.cleanup()
```

The example includes a compact random forest we have trained on nudging tendencies towards reanalysis data. The separation of physics and dynamics steps in the code makes it clear that the machine learning update is applied at the end of a physics step, and is included in any intermediate restart data. The random forest model used in this example is trained according to the approach in Watt-Meyer et al. (2021), with a small number of trees and layers chosen to decrease model size. As a proof of concept, the example model has not been tuned for stability, and may crash if run for longer than 6 hours or using a run directory other than the example provided. Model stability can be increased by enforcing the model specific humidity to be non-negative after applying the random forest update.

This example showcases how the wrapper makes it easy to modify the operation of the Fortran model. In our own efforts to re-write the FV3 dynamical core in a Python-based domain-specific language (DSL), we have directly replaced a call to the Fortran dynamics step with Python-based code. We have also added nudging routines that directly access Zarr (Miles et al., 2020) reference datasets stored in the cloud, and I/O routines to save model snapshots to Zarr files in cloud storage as the model executes. With Python's threading support, this data transfer can happen as the Fortran code is running. These tasks would be difficult to implement in Fortran, due to more complex threading interfaces, no existing bindings for Zarr, and a lack of support from cloud storage providers.

3.2 MPI communication

When writing parallel models, inter-process communication is an important functionality. MPI4py (Dalcín et al., 2008) provides Python bindings for MPI routines, and supports the use of numpy arrays. Using MPI4py, we have been able to implement halo updates, gather, and scatter operations. The syntax for MPI4py is similar to the syntax used in Fortran. In our implementation, the same MPI communicator is used by the Fortran code as is used by MPI4py.

Here we show a simple example of computing the minimum global surface temperature and printing it from the root process. This showcases how you can use MPI4py within the model to compute diagnostics using inter-rank communication.

```python
import fv3gfs.wrapper
import numpy as np
from mpi4py import MPI

ROOT = 0

if __name__ == "__main__":
    fv3gfs.wrapper.initialize()
    # MPI4py requires a receive "buffer" array to store incoming data
    min_surface_temperature = np.array(0)
    for i in range(fv3gfs.wrapper.get_step_count()):
        fv3gfs.wrapper.step_dynamics()
        fv3gfs.wrapper.step_physics()

        # Retrieve model minimum surface temperature
        state = fv3gfs.wrapper.get_state(["surface_temperature"])
        MPI.COMM_WORLD.Reduce(
            state["surface_temperature"].view[:].min(),
            min_surface_temperature,
            root=ROOT,
            op=MPI.MIN,
        )
        if MPI.COMM_WORLD.Get_rank() == ROOT:
            units = state["surface_temperature"].units
            print(f"Minimum surface temperature: {min_surface_temperature} {units}")

        fv3gfs.wrapper.save_intermediate_restart_if_enabled()
    fv3gfs.wrapper.cleanup()
```

3.3 Interactive use in a Jupyter notebook

While we typically run the model using batch submission or from the command line, all of the examples above can be executed from within a Jupyter notebook using ipyparallel. This allows retrieving, computing, and plotting variables from the Fortran

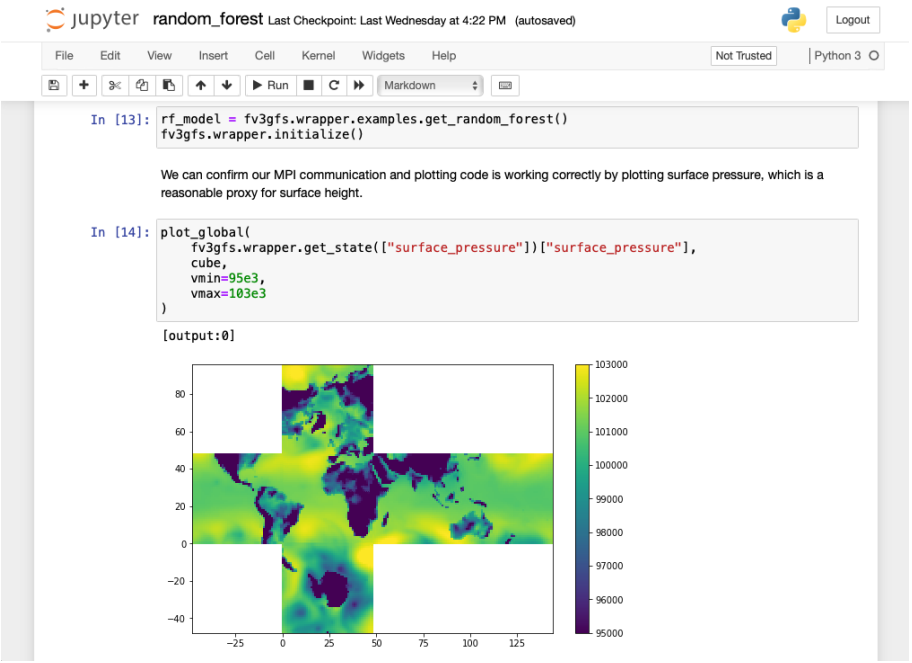

Figure 3. Screenshot of Jupyter notebook example using MPI communication to gather a field on 6 processes and plot it on the first process. Note the FV3GFS uses a "cubed sphere" grid, so that for 6 processes each one is responsible for one face of the cube.

model while it is paused at a point of interest. It also can serve as explicit documentation of modelled phenomena, whether to communicate to other model developers or for use in an educational setting.

We have prepared an example that inspects the machine learning example model, using MPI communication from Python to gather and plot variables on a single rank (Figure 3). It can be accessed in the `examples/jupyter` directory of the Github repository, and makes use of Docker to ensure portability. While the example is written to run on 6 processes, ipyparallel allows notebooks to be run at larger scales on high-performance computing (HPC) clusters if the configuration is modified appropriately.

4 Implementation

4.1 Information transfer

To augment the Fortran model, we must read from and write to its state. This information transfer can be done in two ways, either by providing an interface to copy data between Fortran and Python arrays (effectively C arrays), or using the same memory in both codes. Re-using memory requires that the Fortran code use a pointer to a C array, allowing the same pointer to be used by the numpy array on the Python side. In the Fortran code for FV3GFS, arrays used for physical variables are defined as non-target `allocatable` arrays, which precludes sharing them with Python. It would require significant changes to the

Fortran code to instead use pointers to C arrays, which conflicts with our priority of making minimal changes to the Fortran code. Instead, the getters and setters (the routines which transfer variable values between Python and Fortran) perform a data copy between numpy-allocated C arrays and Fortran arrays within the wrapper layer.

4.2 Metaprogramming to pass arrays

Copying data from Python into a Fortran array requires a significant amount of code. Unless a structure of the Fortran data can be assumed, each Fortran variable to be accessed needs at least its own line of Fortran code, and in practice its own pair of subroutines, containing an assignment between a Python buffer and the correct Fortran variable. In our approach, each variable has two Fortran wrapper subroutines for getting and setting that variable, logic within a C wrapper layer for calling those Fortran wrapper subroutines, and header declarations for those subroutines.

Writing each of these manually would take significant time and effort. Instead we use Jinja templates (The Pallets Projects, 2019) to generate these wrappers using JSON files declaring necessary information such as the Fortran variable name, standard name, units, and dimensionality. For example, the Fortran variable name "zorl" has standard name "surface_roughness", units "cm", and dimensionality `["y", "x"]`. This greatly reduces the number of lines required to write the code. For physics variables, the template file and data file are 89 and 459 lines, respectively while the generated Fortran file is 1894 lines. Physics variables are also responsible for most of the lines in the 1680-line generated Cython file. Adding a new physics variable requires adding an entry to a JSON file with its standard name, Fortran name, Fortran container struct name, dimensions, and units. This JSON file is also used to automatically enable unit tests for the getters and setters of each physics variable.

4.3 Portability and testing using Docker

One choice made in developing fv3gfs-wrapper was to use Docker containers for testing and our own research use of the wrapped model. Using a Docker image ensures that across systems, we can consistently install the dependencies of FV3GFS, Flexible Modeling System (FMS), and Earth System Modeling Framework (ESMF) on a host system. Users can make use of the Docker container identically on cloud computing resources, continuous integration systems, and our host machines without the need for separate configuration of the compilation process for each system. This removes the possibility for error from incorrect build instructions or execution of those instructions, or unexpected interactions with the host environment. Furthermore, it documents the process required to build the environment for the model and fv3gfs-wrapper, which should help in setting it up directly (i.e. without use of containers) on a machine. Finally, it facilitates distribution of the model to others who may not have access to HPC resources and may want to reproduce our results on personal computers or cloud resources. The docker image at time of publication can be retrieved as `gcr.io/vcm-ml/fv3gfs-wrapper:v0.6.0`, or from McGibbon et al. (2021b).

4.4 Extending this approach to other models

While we have applied this wrapping approach to the FV3GFS model specifically, nothing about it is particular to this model. Our methodology should generalize to other atmospheric and climate models. This wrapper is an example of how one can wrap Fortran models in general to be accessible through Python. While using Cython and Fortran wrapper layers (as we have done here) involves writing more code than using automated wrapping tools such as f90wrap (Kermode, 2020), it provides the flexibility necessary to wrap the existing Fortran code with minimal changes. We found much of the repetitive boilerplate needed for this wrapping could be handled through Jinja templating. With this approach, a Python wrapper can be produced for very complex build systems with only minimal modifications (such as ensuring the necessary variables and routines are externally accessible) to the existing model code.

The use of getters and setters introduces a copy overhead cost when modifying the base model behavior. However, it avoids refactoring necessary for a shared memory implementation, which would require modifying the Fortran code to use C-accessible arrays that can be shared with Python. Writing a Fortran wrapper layer for the getters and setters ensures that any variable modifiable in Fortran can also be modified in Python.

In wrapping the FV3GFS, we have split the FV3GFS model main loop into a sequence of subroutines, which are then wrapped to call from Python. This task is likely to be different in other Fortran models, particularly models with abstract main loops or complex coupling infrastructures. So long as Fortran subroutines can be defined to execute each part of the model main loop, these can be wrapped to call from Python for model integration.

4.5 Challenges and limitations

Python reads many files on initialization when it imports packages. This can cause significant slow-down on HPC systems using shared filesystems. Approaches using parallel filesystems, such as Sarus on HPC or Docker-based cloud solutions, can avoid this issue. When a shared filesystem must be used, solutions exist such as python-mpi-bcast by Yu Feng (Feng, 2021), or modifying the CPython binary as reported by Enkovaara et al. (2011).

The wrapper currently treats the dynamics and physics routines each as a single subroutine. This does not allow inserting Python code within the physics suite, between schemes. This limitation may be removed in the future by adding a wrapper for physics schemes in the Common Community Physics Package (CCPP, Heinzeller et al. (2020)). Through CCPP, it should be possible to separate the physics driver into multiple calls, allowing Python code to be called between any chosen physics schemes.

It is also important to remember when trying to modify the behavior of FV3GFS that, with or without a wrapper, it is still fundamentally a complex parallel model. Parallel code is difficult to test, since the order of code execution is non-deterministic (Bianchi et al., 2018).

It may also be necessary to understand the physical relationships between different model variables in the Fortran code. For example, the dry air mass of a model layer in FV3GFS is a diagnostic function of the layer pressure thickness and tracer mixing ratios. Increasing the model specific humidity will remove dry air mass, unless the layer pressure thickness is also increased.

To account for this, we have included a routine `fv3gfs.set_state_mass_conserving` that modifies layer pressure thickness according to any changes in water tracer amounts.

5 Conclusions

We have presented `fv3gfs-wrapper`, a Python-wrapped version of the FV3GFS atmospheric model. The wrapper allows users to control and interact with an atmospheric model written in Fortran. The simple and intuitive interface allows for a Python-centric workflow and can be used to enable a wide range of use cases, such as machine learning parameterization development, online analysis, and interactive model execution. We do not see a decrease in model performance relative to the fully-compiled model, unless routines to copy the model state in and out of the Fortran model are used. This copy overhead is well within an acceptable range of performance, and could be avoided with modifications to the Fortran source code.

We showed examples of how Python and Docker can be used to reproduce and modify the existing Fortran model, and how the Fortran code can be called in an interactive Jupyter environment. In addition to accelerating research and development workflows, these examples show how a full-fledged weather and climate model can be made available for reproducible science and teaching.

The wrapping approach that is outlined can be applied similarly to other Fortran models. The Python-wrapped FV3GFS atmospheric model shows the way for a new generation of weather and climate models, where the top-level control flow of the model is implemented in a high-level language such as Python while the performance critical sections are implemented in a low-level, performant language. This is a powerful approach that has already been used in popular Python packages such as Numpy and Tensorflow. We hope to see this approach extended to other models, enabling more widespread access to Python tools in developing traditional Fortran models, and reducing the barrier to access for researchers and students interested in introducing online analysis code into these models.

Code and data availability. Code for this project is available in on GitHub at https://github.com/VulcanClimateModeling/fv3gfs-wrapper tag v0.6.0 (McGibbon et al., 2021a), https://github.com/VulcanClimateModeling/fv3gfs-fortran tag gmd_submission (Heinzeller et al., 2021), and https://github.com/VulcanClimateModeling/fv3gfs-util tag v0.6.0 (McGibbon et al., 2021c). It is also available as a Docker image at gcr.io/vcm-ml/fv3gfs-wrapper:v0.6.0 (McGibbon et al., 2021b). The model forcing directory used to time the examples is available as McGibbon et al. (2021d).

Author contributions. Jeremy McGibbon contributed the initial version of the wrapper and has led its development. Significant code contributions have been made by Noah Brenowitz, Oliver Watt-Meyer, Spencer Clark, Mark Cheeseman, Brian Henn, Tobias Wicky, Oliver Fuhrer, and Anna Kwa. All authors were involved in design discussions and provided feedback on the code. Jeremy McGibbon prepared the manuscript with contributions from co-authors.

Competing interests. The authors declare that they have no conflict of interest.

Acknowledgements. We thank Vulcan, Inc. for supporting this work. We acknowledge NOAA-EMC, NOAA-GFDL and the UFS Community for publicly hosting source code for the FV3GFS model (https://github.com/ufs-community/ufs-weather-model) and NOAA-EMC for providing the necessary forcing data to run FV3GFS. The FV3GFS model code used was forked from the UFS public release branch in December 2019. Computations supporting this work were also supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID s1053.

References

Bianchi, F. A., Margara, A., and Pezzè, M.: A Survey of Recent Trends in Testing Concurrent Software Systems, IEEE Transactions on Software Engineering, 44, 747–783, https://doi.org/10.1109/TSE.2017.2707089, 2018.

Brenowitz, N. D. and Bretherton, C. S.: Spatially Extended Tests of a Neural Network Parametrization Trained by Coarse-Graining, Journal of Advances in Modeling Earth Systems, 11, 2728–2744, https://doi.org/10.1029/2019MS001711, 2019.

Curcic, M.: A parallel Fortran framework for neural networks and deep learning, CoRR, abs/1902.06714, 2019.

Dalcín, L., Paz, R., Storti, M., and D'Elía, J.: MPI for Python: Performance improvements and MPI-2 extensions, Journal of Parallel and Distributed Computing, 68, 655–662, https://doi.org/10.1016/j.jpdc.2007.09.005, 2008.

Enkovaara, J., Romero, N. A., Shende, S., and Mortensen, J. J.: GPAW - massively parallel electronic structure calculations with Python-based software, Procedia Computer Science, 4, 17–25, https://doi.org/10.1016/j.procs.2011.04.003, 2011.

Feng, Y.: python-mpi-bcast, https://github.com/rainwoodman/python-mpi-bcast, 2021, (accessed: April 7, 2021).

Global Engineering Documents: Fortran 90, Global Engineering Documents, Washington, DC, USA, 1991.

Hand, N. and Feng, Y.: nbodykit, in: Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing - PyHPC'17, ACM Press, https://doi.org/10.1145/3149869.3149876, https://doi.org/10.1145/3149869.3149876, 2017.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E.: Array programming with NumPy, Nature, 585, 357–362, https://doi.org/10.1038/s41586-020-2649-2, 2020.

Harris, L. M. and Lin, S.-J.: A Two-Way Nested Global-Regional Dynamical Core on the Cubed-Sphere Grid, Monthly Weather Review, 141, 283 – 306, https://doi.org/10.1175/MWR-D-11-00201.1, 2013.

Heinzeller, D., Firl, G., Bernardet, L., Carson, L., Zhang, M., and Kain, J.: The Common Community Physics Package (CCPP): bridging the gap between research and operations to improve U.S. numerical weather prediction capabilities, EGU General Assembly 2020, https://doi.org/10.5194/egusphere-egu2020-23, 2020.

Heinzeller, D., Underwood, S., Firl, G., Wang, J., Liang, Z., Menzel, R., Robinson, T., Brown, T., Benson, R., Hartnett, E., Schramm, J., Ramirez, U., gbw gfdl, Carson, L., McGibbon, J., Fuhrer, O., Jess, Clark, S., Smirnova, T., Hallberg, R., Bernardet, L., Potts, M., Zadeh, N., Olson, J., Jovic, D., George, R., Paulot, F., Goldhaber, S., and Li, H.: VulcanClimateModeling/fv3gfs-fortran: GMD release, Zenodo [code], https://doi.org/10.5281/zenodo.4470023, 2021.

Kermode, J. R.: f90wrap: an automated tool for constructing deep Python interfaces to modern Fortran codes, J. Phys. Condens. Matter, https://doi.org/10.1088/1361-648X/ab82d2, 2020.

Lin, J. W.-B.: qtcm 0.1.2: a Python implementation of the Neelin-Zeng Quasi-Equilibrium Tropical Circulation Model, Geoscientific Model Development, 2, 1–11, https://doi.org/10.5194/gmd-2-1-2009, 2009.

McGibbon, J., Brenowitz, N. D., Watt-Meyer, O., Clark, S., Cheeseman, M., Henn, B., Fuhrer, O., Wicky, T., and Elbert, O.: VulcanClimateModeling/fv3gfs-wrapper: v0.6.0 GMD release, Zenodo [code], https://doi.org/10.5281/zenodo.4474598, 2021a.

McGibbon, J., Brenowitz, N. D., Watt-Meyer, O., Clark, S., Cheeseman, M., Henn, B., Fuhrer, O., Wicky, T., and Elbert, O.: VulcanClimateModeling/fv3gfs-wrapper: v0.6.0 GMD release Docker Image, Zenodo, https://doi.org/10.5281/zenodo.4474639, 2021b.

McGibbon, J., Watt-Meyer, O., Brenowitz, N. D., Clark, S., Kwa, A., Cheeseman, M., Wicky, T., and Henn, B.: VulcanClimateModeling/fv3gfs-util: v0.6.0 GMD release, Zenodo [code], https://doi.org/10.5281/zenodo.4470011, 2021c.

McGibbon, J., Watt-Meyer, O., Yang, F., and Harris, L.: Example 6-hour directory for FV3GFS atmospheric model, Zenodo, https://doi.org/10.5281/zenodo.4429298, 2021d.

Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 3.1, 2015.

Miles, A., jakirkham, Durant, M., Bussonnier, M., Bourbeau, J., Onalan, T., Hamman, J., Patel, Z., Rocklin, M., shikharsg, Abernathey, R., Moore, J., Schut, V., raphael dussin, de Andrade, E. S., Noyes, C., Jelenak, A., Banihirwe, A., Barnes, C., Sakkis, G., Funke, J., Kelleher, J., Jevnik, J., Swaney, J., Rahul, P. S., Saalfeld, S., john, Tran, T., pyup.io bot, and sbalmer: zarr-developers/zarr-python: v2.5.0, Zenodo [code], https://doi.org/10.5281/zenodo.4069231, 2020.

Monteiro, J. M., McGibbon, J., and Caballero, R.: sympl (v. 0.4.0) and climt (v. 0.15.3) – towards a flexible framework for building model hierarchies in Python, Geoscientific Model Development, 11, 3781–3794, https://doi.org/10.5194/gmd-11-3781-2018, 2018.

OpenMP Architecture Review Board: OpenMP Application Programming Interface Version 5.0, 2020.

Ott, J., Pritchard, M., Best, N., Linstead, E., Curcic, M., and Baldi, P.: A Fortran-Keras Deep Learning Bridge for Scientific Computing, Scientific Programming, 2020, 8888 811, https://doi.org/10.1155/2020/8888811, 2020.

Peterson, P.: F2PY: a tool for connecting Fortran and Python programs, International Journal of Computational Science and Engineering, 4, 296, https://doi.org/10.1504/ijcse.2009.029165, 2009.

Pressel, K. G., Kaul, C. M., Schneider, T., Tan, Z., and Mishra, S.: Large-eddy simulation in an anelastic framework with closed water and entropy balances, Journal of Advances in Modeling Earth Systems, 7, 1425–1456, https://doi.org/10.1002/2015MS000496, 2015.

Putman, W. M. and Lin, S.-J.: Finite-volume transport on various cubed-sphere grids, Journal of Computational Physics, 227, 55 – 78, https://doi.org/https://doi.org/10.1016/j.jcp.2007.07.022, 2007.

The Pallets Projects: Jinja, https://jinja.palletsprojects.com/en/2.10.x/, 2019.

Watt-Meyer, O., Brenowitz, N. D., Clark, S. K., Henn, B., Kwa, A., McGibbon, J. J., Perkins, W. A., and Bretherton, C. S.: Correcting weather and climate models by machine learning nudged historical simulations, Earth and Space Science Open Archive, p. 13, https://doi.org/10.1002/essoar.10505959.1, 2021.

Zhou, L., Lin, S.-J., Chen, J.-H., Harris, L. M., Chen, X., and Rees, S. L.: Toward Convective-Scale Prediction within the Next Generation Global Prediction System, Bulletin of the American Meteorological Society, 100, 1225 – 1243, https://doi.org/10.1175/BAMS-D-17-0246.1, 2019.