A Python interface to the Fortran-based Parallel Data Assimilation Framework: pyPDAF v1.0.2

Yumeng Chen[1,2], Lars Nerger[3], and Amos S. Lawless[1,2]

[1]School of Mathematical, Physical and Computational Sciences, University of Reading, Reading RG6 6ET, UK
[2]National Centre for Earth Observation, University of Reading, Reading RG6 6ET, UK
[3]Alfred-Wegener-Institut, Helmholtz-Zentrum für Polar-und Meeresforschung (AWI), 27570 Bremerhaven, Germany

Correspondence: Yumeng Chen (yumeng.chen@reading.ac.uk)

Abstract. Data assimilation (DA) is an essential component of numerical weather and climate prediction. Efficient implementation of DA algorithms benefits both research and operational prediction. Currently, a variety of DA software programs are available. One of the notable DA libraries is the Parallel Data Assimilation Framework (PDAF) designed for ensemble data assimilation. The DA framework is widely used with complex high-dimensional climate models, and is applied for research on atmosphere, ocean, sea ice and marine ecosystem modelling, as well as operational ocean forecasting. Meanwhile, there ~~exists increasing need~~ are increasing demands for flexible and efficient DA implementations using Python due to the increasing amount of intermediate complexity models as well as machine learning based models coded in Python. To accommodate for such ~~needs~~demands, we introduce a Python interface to PDAF, pyPDAF. pyPDAF allows for flexible DA system development while retaining the efficient implementation of the core DA algorithms in the Fortran-based PDAF. The ideal use-case of pyPDAF is a DA system where the model integration is independent from the DA program, which reads the model forecast ensemble, produces ~~a model analysis~~an analysis, and updates the restart files of the model, or a DA system where the model can be used in Python. With implementations of both PDAF and pyPDAF, this study demonstrates the use of pyPDAF and PDAF ~~for~~ in a coupled data assimilation (CDA) setup in a coupled atmosphere-ocean model, the Modular Arbitrary-Order Ocean-Atmosphere Model (MAOOAM). This study demonstrates that pyPDAF allows for ~~the utilisation of Python user-supplied functions with PDAF functionalities~~ PDAF functionalities from Python where users can utilise Python functions to handle case-specific information from observations and numerical model. The study also shows that pyPDAF can be used with high-dimensional systems with little slow-down per analysis step of only up to 13% for the localized ensemble Kalman filter LETKF ~~. In addition, our CDA experiments confirm the benefit of strongly coupled data assimilation for improving both the instantaneous state and the long-term trend of the coupled dynamical system.~~ in the example used in this study. The study also shows that, compared to PDAF, the overhead of pyPDAF is comparatively smaller when computationally intensive components dominate the DA system. This can be the case for systems with high-dimensional state vectors.

1 Introduction

Data assimilation (DA) ~~is widely used in weather and climate modelling where observations are used to constrain the model prediction based on the uncertainty of both the observations and the model forecast~~combines simulations and observations

using dynamical systems theory and statistical methods. This process provides optimal estimates (i.e., analyses), enables parameter estimation, and allows for the evaluation of observation networks. Due to the limited predictability and imperfect models, DA has become one of the most important techniques for ~~the~~ numerical weather and climate predictions. ~~Progresses~~ Progress of the DA methodology development can be found in various review articles and books (e.g., Bannister, 2017; Carrassi et al., 2018; Vetra-Carvalho et al., 2018; Evensen et al., 2022).

To ameliorate the difficulties in the implementation of different DA approaches, several DA software programs and libraries have been proposed (e.g., Nerger et al., 2005; Anderson et al., 2009; Raanes et al., 2024; Trémolet and Auligne, 2020). Even though the implementation of the core DA algorithms is similar, these software programs/libraries are typically tailored to different purposes. For example, the Joint Effort for Data assimilation Integration (JEDI, Trémolet and Auligne, 2020) is a piece of self-contained software that includes a variety of functionalities that can be used for all aspects of a DA system mainly for operational purposes while DA software for methodology research such as ~~DAPPER (Raanes et al., 2024)~~ Data Assimilation with Python: a Package for Experimental Research (DAPPER, Raanes et al., 2024) is designed for identical twin experiments equipped with low complexity models.

One widely used DA framework is the Parallel Data Assimilation Framework (PDAF) developed and maintained by the Alfred Wegener Institute (Nerger et al., 2005; Nerger and Hiller, 2013b). The framework is designed for efficient ~~implementations~~ implementation of ensemble-based DA systems for complex weather and climate models but is also used for research on ~~data assimilation~~ DA methods with low-dimensional "toy" models. ~~The DA implementations require user-supplied functions to provide~~ In this generic framework, DA methods accommodate case-specific information about the DA system ~~including the~~ through functions provided by users including the model fields, treatment of observations, and localisation. These functions are referred to as *user-supplied functions*. More than 100 studies have used PDAF, including atmosphere (e.g., Shao and Nerger, 2024), ocean (e.g., Losa et al., 2012; Pohlmann et al., 2023), sea ice (e.g., Williams et al., 2023; Zhao et al., 2024), land surface (e.g., Strebel et al., 2022; Kurtz et al., 2016), hydrology (e.g., Tang et al., 2024; Döll et al., 2024), and coupled systems ~~(e.g., Nerger et al., 2020)~~(e.g., AWI-CM in Nerger et al., 2020). Further use-cases of PDAF can be found in the PDAF website (PDAF - the Parallel Data Assimilation Framework, last access: 2024-02-13). Even though PDAF provides highly optimised DA algorithms, the flexible framework relies on the user-supplied functions to couple DA with model system and observations. The implementation of user-supplied functions still require additional code development, which can be time-consuming especially when the routines have to be written in Fortran, a popular programming language for weather and climate applications.

In recent years, Python is gaining popularity in weather and climate communities due to its flexibility and ease of implementation. For example, Python is adopted by some low- to intermediate-complexity models (e.g., De Cruz et al., 2016; Abernathey et al., 2022), models with a Python wrapper (e.g., McGibbon et al., 2021), and machine learning based models (e.g., Kurth et al., 2023; Lam et al., 2023; Bi et al., 2023). For the application of DA in Python, DAPPER provides a variety of DA algorithms for twin experiments using low-dimensional Python models. The Ensemble and Assimilation Tool, EAT (Bruggeman et al., 2024)~~was proposed to set up a 1D ocean-biogeochemical DA system~~, which is a wrapper to a Fortran data assimilation system based on PDAF, was proposed to set up a 1D ocean-biogeochemical DA system including the 1D ocean-biogeochemical model,

GOTM-FABM. There are also Python packages designed mainly for pedagogical purposes in low-dimensional systems such as openDA (Ahmed et al., 2020) and filterpy (filterpy PyPI, last access: 2024-08-29). For high-dimensional applications, there are efficient implementations of DA packages such as HIPPYlib by Villa et al. (2021) and ADAO (SALOME The Open Source Integration Platform for Numerical Simulation, last access: 2024-08-29), but HIPPYlib does not have a focus on ensemble data assimilation approaches whereas ADAO provides various ensemble DA methodologies but it has no support for the localisation used in weather and climate applications. More recently, NEDAS (Ying, 2024) was introduced for offline ensemble DA in climate applications but it currently only supports limited DA algorithms.

Targeted at applications to high-dimensional ensemble data assimilation systems, here, we introduce a Python interface to PDAF, pyPDAF. Using pyPDAF, one can implement both offline and online DA systems using Python. For offline DA systems, DA is performed utilising files written onto a disk, e.g., model restart files. If a numerical model is available in Python, pyPDAF allows for implementing an online DA system ~~implementation~~ where DA algorithms can be used with the Python model with in-memory data exchange that does not need I/O operations bringing about more efficiency than an offline system. Compared to user-supplied functions implemented in Fortran, the Python implementation can facilitate easier code development thanks to a variety of packages readily available in Python. In the meantime, DA algorithms provided by PDAF that are efficiently implemented in Fortran can still be utilised.

In this study, we ~~demonstrate the use of pyPDAFin a~~ introduce the design, implementation and functionalities of pyPDAF. Further, in comparison to the existing PDAF implementation, we provide a use-case of pyPDAF in a coupled data assimilation (CDA) setup with the Modular Arbitrary-Order Ocean-Atmosphere Model (MAOOAM, De Cruz et al., 2016) where an arbitrary number of grid points can be specified without changing the model dynamics making it suitable to provide benchmarks of pyPDAF. ~~The research on CDA is motivated by the use of coupled earth system models, especially for coupled atmosphere and ocean simulations (Eyring et al., 2016; Walters et al., 2019). Traditionally, each model component is assimilated individually and the state of each model component interacts with the others only in the coupled model forecast. This approach is called weakly coupled DA~~ This use-case allows us to further compare the ease of implementation of pyPDAF with PDAF, and investigate the computational performance under different choices of state vector including both the strongly CDA (SCDA) and weakly CDA (WCDA) ~~. It is desirable to perform DA jointly for all model components simultaneously, usually denoted as strongly coupled DA (SCDA). Studies report a suite of benefits of using SCDA. For example, Smith et al. (2015) shows that the SCDA can improve dynamical balance in the analysis leading to reduced initialisation shocks. Sluka et al. (2016) reported improvements in analysis with SCDA in an intermediate complexity model. Tang et al. (2021) performed SCDA of ocean observations into the coupled atmosphere-ocean model AWI-CM and found positive effects in particular in the polar regions. Further studies can be found in a suite of review articles on CDA (Penny and Hamill, 2017; Zhang et al., 2020; de Rosnay et al., 2022; Kaln~~ . cases. In the SCDA case, the atmosphere and ocean are coupled in both forecast and analysis steps. In the WCDA case, the forecast step is coupled but the analysis of the atmosphere and ocean is performed independently.

Here, ~~we will first introduce ensemble-based data assimilation, the principal objective of PDAF, in~~ Sect. ~~2.2. Section 2 will describe the design and implementation of PDAF and~~ 2 will introduce PDAF, the implementation and design of pyPDAF, and

the implementation of a DA system in pyPDAF. In Sect. 3, the experimental and model setup will be described. Section 4 will report the performance of PDAF and pyPDAF in the CDA setup. We will conclude in Sect. 5.

2 Ensemble-based data assimilation

Although PDAF supports a few deterministic DA methods, it focuses on ensemble-based DA methods. Ensemble-based DA is a class of DA approaches that approximate the statistics of the model state and its uncertainty using an ensemble of model realisations motivated by DA approaches based on Bayes theorem where the prior, typically a model forecast, and posterior (analysis) distributions can be approximated by a Monte Carlo approach. The ensemble model forecast allows for an embarrassingly parallel implementation which means that, with sufficient computational resources, the wall clock computational time of the forecast does not increase with the ensemble size.

Under the Gaussian assumption of the forecast and analysis distributions, one of the most notable ensemble-based DA methods is the ensemble Kalman filter (EnKF, Evensen, 1994). The EnKF approximates the forecast and analysis error distribution by an ensemble. The method was proven to be successful in many applications (e.g., Houtekamer et al., 2005; Feng et al., 2009; Hamill et a
. To further improve the efficiency and reliability of the EnKF, multiple variants of the EnKF were proposed, such as singular evolutive intepolated Kalman filter (SEIK, Pham, 2001), ensemble transform Kalman filter (ETKF, Bishop et al., 2001), error space transform Kalman filter (ESTKF, Nerger et al., 2012), and the deterministic ensemble Kalman filter (Sakov and Oke, 2008)
. In practice computational resources limit the feasible ensemble size, which is typically of an order of 10 to 100, in the high-dimensional realistic DA applications in the Earth system due to the cost of model forecasts. The ensemble-based DA approaches typically suffer from sampling errors from limited ensemble size. To counter these deficiencies, covariance matrix inflation and localisation are commonly used (e.g., Pham et al., 1998; Hamill et al., 2001; Hunt et al., 2007). In particular, the domain localisation is tailored for efficient parallel implementations that are commonly used in high-dimensional DA systems.

Ensemble-based DA can also treat fully non-linear non-Gaussian problems. The most notable example is particle filters (see, van Leeuwe
. They can be used to solve fully non-linear problems without assumptions on the prior and posterior distribution. However, for high-dimensional geoscience applications, the classical particle filters suffer from the "curse of dimensionality" where the required ensemble size grows exponentially with the dimension of the state vector making the approach computationally infeasible. Recent developments of the particle filters significantly improve the stability and reduce the required ensemble size of the approach making it a potential choice for low-to-medium complexity models, such as implicit equal-weights particle filters (Zhu et al., 2016) and the particle flow filter (Hu and van Leeuwen, 2021). An overview of other developments of particle filters can be found in van Leeuwen et al. (2019).

The ensemble-based DA approaches are adopted by many operational centres where traditionally variational methods are used (e.g., Clayton et al., 2013; Caron et al., 2015; Bonavita et al., 2016; Hersbach et al., 2020). In variational methods, ensemble approaches are used to achieve flow-dependent background covariance matrix, and/or to avoid explicit computation of the

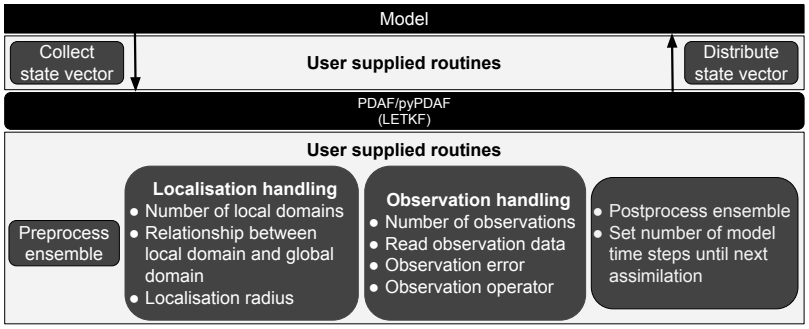

Figure 1. A schematic diagram of an online LETKF DA system using (py)PDAF. In the case of an offline DA system, the model can be its restart files.

~~adjoint model in the minimisation process by using an ensemble approximation. These goals can be realised using various different methodologies and a detailed review of these methods can be found in Bannister (2017).~~

2 PDAF and PyPDAF

PDAF is designed for research and operational DA systems. As a Python interface to PDAF, pyPDAF inherits the DA algorithms implemented in PDAF and the same implementation approach to build a DA system.

2.1 Parallel Data Assimilation Framework (PDAF)

PDAF is a Fortran-based DA framework providing fully optimised, parallelised ensemble-based DA algorithms. The framework provides a software library and defines a suite of workflows based on different DA algorithms provided by PDAF~~including various ensemble Kalman filters/smoothers, ensemble-based 3DVar (Bannister, 2017), particle filters (van Leeuwen et al., 2019) and other non-linear filters (Tödter and Ahrens, 2015; Nerger, 2022). To deal with sampling errors in the ensemble-based DA , the framework also provides options for adaptive inflation and localisation schemes.~~. The DA algorithms provided by PDAF will be given in Sect. 2.2.

~~As a framework for ensemble DA, it comes with the functionality to generate the initial ensemble. One possibility is to use the second-order exact sampling (Pham, 2001) where the ensemble is generated based on the model trajectory of the modelled truth. The assumption is that the uncertainty of the model initial condition lies in the phase space of the model trajectory. The space is represented by the singular values and its corresponding vectors using an empirical orthogonal function (EOF) decomposition.~~

To ensure that PDAF can be flexibly adapted to any models and observations, it requires users to provide case-specific information. This includes the information on the state vector, observations and localisation. The framework obtains this information via *user-supplied functions* which are external callback subroutines. Figure 1 shows a schematic diagram of an online DA system where the ~~LETKF~~ local ensemble transform Kalman filter (LETKF) is used. Here, the user-supplied functions con-

nect PDAF with models. Called ~~within~~ by the PDAF routines, these user-supplied functions collect state vectors from model forecasts and distribute the analysis back to the model for the following forecast phase. During the analysis step, user-supplied functions also pre- and post-process the ensemble, handle ~~localisations~~ localisation and observations, and provide the number of model time steps for the ~~next forecast phaseto PDAF~~following forecast phase. As the user-supplied functions depend on the chosen DA algorithm, other algorithms may require different functions. For example, ~~the~~ a 3DVar DA system requires routines for the adjoint observation operator and control vector transformation. To ameliorate the difficulty in the observation handling, PDAF provides a scheme called observation module infrastructure (OMI). The OMI routines provide a structured way to handle the processing of ~~observation vectors~~ observations and error covariance matrix used by DA algorithms, and provide support for the complex distance computation used by localisation. In the current version of PDAF V2.3, it also supports spatial interpolations on structured and unstructured grids ~~, direct observation operator, and a diagonal or~~ for observation operators as well as an observation operator for observations located on grid points. The OMI also supports both diagonal and non-diagonal observation error covariance ~~matrix~~matrices. One can also implement PDAF without OMI, but additional functions would be required.

In an online DA system, the collection and distribution of state vector is an in-memory data exchange handled ~~by PDAF efficiently~~efficiently by PDAF. It is possible to implement an offline DA system with PDAF where the model in Fig. 1 ~~would be~~ is replaced by model restart files while the user-supplied collection and distribution routines manage the I/O operations of these restart files. Offline DA implementation is a crucially supported feature of PDAF and a potentially important use-case for pyPDAF, but we will not discuss it in detail for the sake of brevity. We will provide details of the ~~use~~ implementation of user-supplied functions in the context of pyPDAF in Sect. 2.4.

2.2 Data assimilation methods in PDAF

As described in Sect. 2, PDAF supports a variety of DA methods with a focus on ensemble-based DA methods. Ensemble-based DA is a class of DA approaches that approximate the statistics of the model state and its uncertainty using an ensemble of model realisations. These DA methods are based on Bayes theorem where the prior, typically a model forecast, and posterior (analysis) distributions can be approximated by a Monte Carlo approach. The ensemble forecast allows for an embarrassingly parallel implementation which means that, with sufficient computational resources, the wall clock computational time of the forecast does not increase with the ensemble size.

The majority of ensemble DA methods are constructed under the Gaussian assumption of the forecast and analysis distributions such as the stochastic ensemble Kalman filter (EnKF, Evensen, 1994). The EnKF approximates the forecast and analysis error distribution by an ensemble. PDAF provides implementations for the EnKF and several of its variants. These variants improve the efficiency and reliability of the EnKF including singular evolutive intepolated Kalman filter (SEIK, Pham, 2001), ensemble transform Kalman filter (ETKF, Bishop et al., 2001), error space transform Kalman filter (ESTKF, Nerger et al., 2012). Other typical filtering algorithms, not implemented in current releases, such as ensemble adjustment Kalman filter (EAKF, Anderson, 2001) and ensemble square root filters (EnSRF, Whitaker and Hamill, 2002) are planned to be included in future releases. In practice, computational resources limit the feasible ensemble size in the high-dimensional realistic DA applications in the Earth system

due to the cost of model forecasts. The ensemble-based DA approaches typically suffer from sampling errors from limited ensemble size. To mitigate these deficiencies, PDAF also provides common techniques such as covariance matrix inflation and localisation (e.g., Pham et al., 1998; Hamill et al., 2001; Hunt et al., 2007). In addition to EnKFs, PDAF also provides 3-dimensional variational methods. This includes variants of 3DEnVar (see Bannister, 2017) that can be used to achieve flow-dependent background error covariance matrix, and/or to avoid explicit computation of the adjoint model in the minimisation process by using an ensemble approximation.

In additional, PDAF also provides DA methods that can treat fully non-linear and non-Gaussian problems. This includes particle filters (see van Leeuwen et al., 2019). However, for high-dimensional geoscience applications, the classical particle filters suffer from the "curse of dimensionality" where the required ensemble size grows exponentially with the dimension of the state vector making the approach computationally infeasible. Therefore, PDAF also provides other non-linear filters such as nonlinear ensemble transform filter (NETF) and local Kalman–nonlinear ensemble transform filter (LKNETF, Tödter and Ahrens, 2015; Ne . These methods mitigate the cost of nonlinear filters by restricting to the second-order moment of the statistical distribution.

2.3 pyPDAF

~~Implementation~~ Depending on the users' programming skills, implementation of user-supplied functions can be laborious in Fortran and typical code development in Python can be less time consuming. Thanks to the integrated package management, code development in Python can rely on well optimised packages without the need for compilation. For these reasons, a variety of numerical models are implemented in Python (e.g., De Cruz et al., 2016; Abernathey et al., 2022; McGibbon et al., 2021; Bi et al., 2023). Hence, a Python interface to PDAF allows the design of an online DA system with such Python-based models. These range from low-dimensional toy dynamical systems to high-dimensional weather and climate systems. Compared to a Fortran-coded DA system, a Python DA system can be implemented efficiently with the aid of various external packages and allows for easier modifications without recompilation such that users can focus on scientific problems.

The pyPDAF package can also be applied ~~for~~ to offline DA systems, i.e. coupling the model and data assimilation program through restart files. ~~Here~~ In an offline DA system, pyPDAF can be used without the restriction of the programming language of the numerical model. When computation-intensive user-supplied functions are well optimised~~(~~, e.g., using just-in-time (JIT) compilation~~)~~, this could also be used for high-dimensional complex models. Thus, depending on the requirements of the users, ~~an offline DA system can~~ pyPDAF can also be used to prototype a Fortran DA system~~as well~~. The application of pyPDAF in high-dimensional models can also be ~~shown~~ enabled by its support of the parallel features of PDAF, which use the Message Passing Interface (MPI, Message Passing Interface Forum, 2023). For this, a pyPDAF DA system relies on the "mpi4py" package for MPI support. The pyPDAF system can also support shared memory parallelisation for DA algorithms using domain localisation in PDAF when built with OpenMP. However, the efficiency of OpenMP is restricted by the global interpreter lock in Python.

As the reference implementation of Python is based on the C programming language (The Python Language Reference, last access: 2024-02-13), the design of pyPDAF is based on the interoperability between the programming languages of C and Fortran using the *iso_c_binding* module of Fortran. As shown in Fig. 2, ~~the~~ a C interface of PDAF, *PDAFc*, is developed in

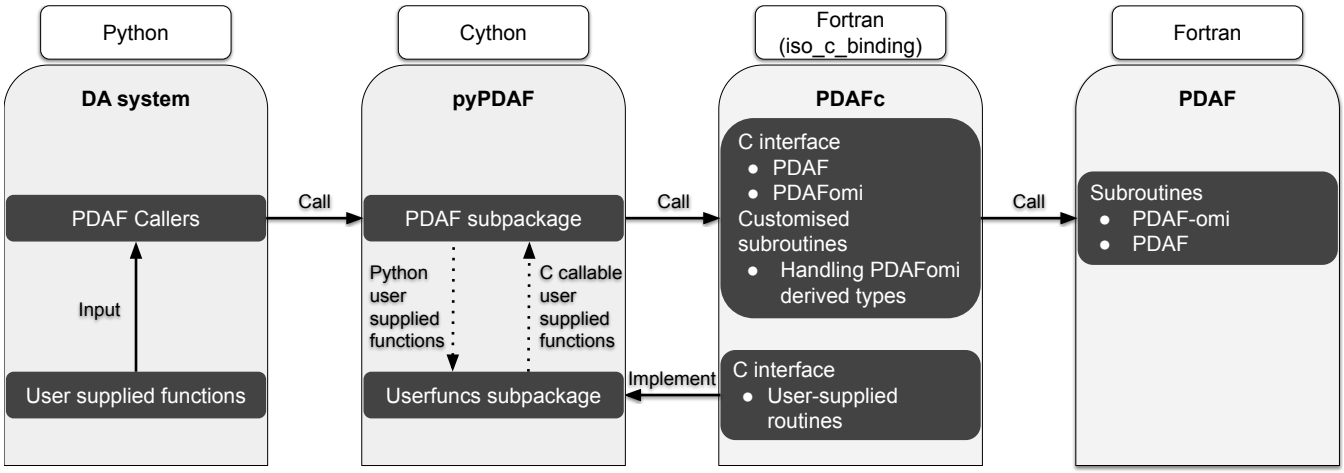

Figure 2. An illustration of the design of the pyPDAF interface to the Fortran-based framework PDAF. Here, only the Python component is exposed to pyPDAF users, and the Cython and Fortran implementations are internal implementations of pyPDAF.

pyPDAF, which includes essential PDAF interfaces and interfaces for user-supplied functions. ~~Hence~~Due to this design, PDAFc could be used ~~independently from pyPDAF as a C interface to the PDAFpackage. The core of the pyPDAF implementation uses~~ as an independent C library for PDAF. The pyPDAF implementation relies on the C-extension for Python (Cython). ~~Here~~ In Cython, Python datatypes are converted into C pointers to allow for information exchange between PDAF and pyPDAF. ~~pyPDAF implements~~ In pyPDAF, C callable functions ~~which can~~ are implemented to call user-supplied functions written in Python such that PDAF can utilise ~~the~~ these user-supplied ~~Python functions.~~ functions. The PDAF functionalities are provided through functions implemented in Cython, which are accessible from Python.

~~pyPDAF is designed so~~ The pyPDAF design means that a DA system can be coded purely in Python ~~including the user-supplied functions and function calls to algorithms implemented in PDAF .~~ utilising PDAF functions. The simplicity of a pure Python DA system depends on mixed programming languages and external libraries. The ~~interface to PDAF is provided through functions implemented using Cython, which provides the interface for calls from Python . Thus, the~~ pyPDAF package itself is a mixed program of C, Fortran and Python. Moreover, as DA algorithms require high-dimensional matrix multiplications, PDAF relies on the numerical libraries LAPACK (linear algebra package) and BLAS (basic linear algebra subprograms). ~~These~~ The mixed languages and libraries lead to a complex compilation process especially when users could use different operating systems. To fully utilise the cross-platform support of Python environment, pyPDAF is distributed via the package manager *conda* to provide an out-of-box user experience with pyPDAF where users can use pyPDAF without the need for compiling the package from the source code. Detailed installation instructions can be found at: Installation - pyPDAF documentation (last access: 2025-03-25) .

pyPDAF allows for the use of efficient implementations of DA algorithms in PDAF. However, a DA system purely based on pyPDAF could still be less efficient than a DA system purely based on PDAF coded in Fortran. The loss of efficiency is partly

due to the operations in user-supplied Python functions and the overhead from the conversion of data types between Fortran and Python. We will evaluate the implications of these loss of efficiency in Sect. 4.2.

2.4 Construction of data assimilation systems using pyPDAF

To illustrate the application of pyPDAF to an existing numerical model, as an example, we present key ~~components~~ implementation details of an LETKF DA system. This example follows the schematic diagram in Fig. 1. Here, we assume that the number of available processors is equal to the ensemble size. ~~In this setup~~Under this assumption, each ensemble member of the model forecast runs on one processor, and the analysis is performed serially on a single processor. We further assume that observations are co-located on the model grid but are of lower resolution, and they have a diagonal error covariance matrix.

~~Compared to Fortran implementations, a Python~~ A pyPDAF DA system can ~~better utilise the object-oriented features. Here, we assume the existence of a generic *model* object that contains model information~~be divided into three components: initialisation, assimilation and finialisation. In this section, to associate the description with actual variable names in pyPDAF, the variable names in pyPDAF are given in brackets. In this ~~sytem~~system, the pyPDAF functionalities ~~should be initialised by~~ are initialised by a single function call

$$param\_int, param\_real, flag = pyPDAF.PDAF.init(filtertype, subtype, stepnull,$$
$$param\_int, param\_real,$$
$$COMM\_model, COMM\_filter, COMM\_couple,$$
$$task\_id, n\_modeltasks, filterpe, init\_ens\_pdaf).$$

~~The information~~ In the initialisation step, the following information is provided to pyPDAF:

1. pyPDAF takes information on the type of filters (*filtertype* and *subtype*)~~is given to PDAF by this function. It also takes parameters of these filters. Here, the~~ .

2. Corresponding to the filter type, the initialisation step also requires the size of the state vector ~~(*dim_p*)~~ and the ensemble size (*dim_p* and *dim_ens* ~~) are specified in the~~ in *param_int* ~~array~~respectively), and the inflation factor ~~is specified in~~ (*param_real*~~array~~). These parameters allow PDAF to allocate internal arrays such as the ensemble mean (*state_p*) and the ensemble matrix (*ens_p*) used by the DA. ~~The MPI communicatorsof model, the filter and the coupling between model and filter are also specified here by~~

3. In addition to the filter configurations, the initialisation step also initialises the parallelisation used in PDAF, which requires the MPI communicators. These MPI communicators instruct PDAF on the functionalities of each processor and their communication patterns. Processors within the same model communicator (*COMM_model*~~, *COMM_filter*, *COMM_couple* respectively. The initialisation function also obtain other parallelisation information from the function call including the index of the parallel model tasks by *task_id*, the total number of parallel model tasks by~~) are used to perform the same ensemble member of model forecast. The number of model communicators is the same as the number

of ensemble members run simultaneously (*n_modeltasks*~~, a boolean variable that determine if the filter~~). Each processor is associated with a specific ensemble member or model task by an index (*task_id*). During the DA step, PDAF will collect an ensemble matrix from the state vector residing in each model communicator via the coupling communicator (*COMM_couple*). The DA is performed on ~~current process by *filterpe*~~filter processors (*filterpe = .true.*) in one of the filter communicators (*COMM_filter*). Even though the parallelisation strategy can be freely designed by users, example parallelisation modules are readily available in pyPDAF. Detailed explanations of the parallelisation strategy used by PDAF can be found in Nerger and Hiller (2013a). ~~Also, the initialisation function~~

4. The initialisation step also prepares the system for future forecast-analysis cycling. Here, it takes the initial time step ~~,~~ (*stepnull*~~, as a step counter~~) for step counters in PDAF.

5. In the initialisation ~~,~~ step, the ensemble has to be initialised. In pyPDAF, this is achieved by the return values of a user-supplied function ~~of~~ (*state_p, uinv, ens_p, flag = init_ens_pdaf(filtertype, dim_p, dim_ens, state_p, uinv, ens_p, flag)*~~is used to initialise PDAF ensemble, *ens_p*. In the user-supplied function, the input arguments are given by the PDAF, and the returned arguments are received by PDAF to perform DA. Here,~~). In this function, users have the flexibility to choose the initial ensemble. One can read an ensemble from files, or sample an ensemble from a covariance matrix. These can be assisted by input from PDAF via arguments of user-supplied functions. For example, PDAF provides the functionality of generating ensemble from singular vectors and values (*uinv*~~is a variable used for the~~) of a covariance matrix using second-order exact sampling ~~. The ensemble generation method can be used with *pyPDAF.PDAF.eofcovar* and *pyPDAF.PDAF.SampleEns* when starting from a deterministic run.~~ (Pham, 2001) .

6. If OMI is used, the initialisation step also involves an additional function call (*pyPDAF.PDAF.omi_init(n_obs)*~~is used to initialise the~~) to inform PDAF the number of observation types (*n_obs*~~types of observations.~~) in the system.

In each model integration step, the ~~analysis step is executed by~~ function

$$status = pyPDAF.PDAF.omi\_assimilate\_local(collect\_state, distribute\_state, init\_dim\_obs,$$
$$obs\_op, prepostprocess, init\_n\_domains,$$
$$init\_dim\_l, init\_dim\_obs\_l,$$
$$g2l\_state, l2g\_state, next\_observation)$$

~~where *status* is a flag for the error code of the DA step, and the arguments of *pyPDAF.PDAF.omi_assimilation_local* are~~ is called where all arguments specify the names of user-supplied ~~functions, which will be discussed in detail.~~ functions handling operations specific to the model and observations. If the forecast phase is complete, the analysis step is executed. In the analysis step, each user-supplied function will next be executed by PDAF to collect necessary information, or perform case-specific operations for the DA. A flow chart is given in Fig 3.

As shown in Fig. 1, the model and PDAF ~~exchanges~~ exchange information by user-supplied functions. ~~The~~ First, a user-supplied function (*state_p = collect_state(dim_p, state_p)*) is executed by PDAF for each ensemble member to ~~fill~~ collect a state

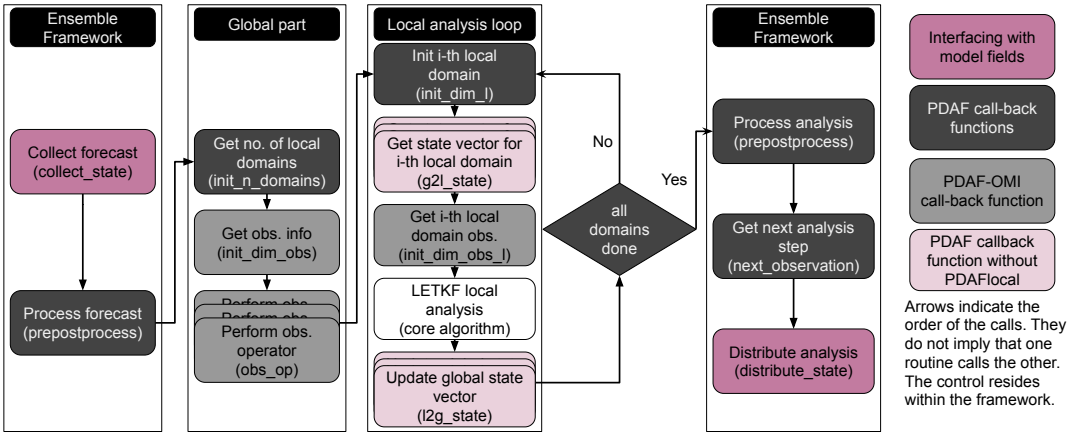

Figure 3. A flowchart of the sequence of LETKF operations in PDAF. These operations include user-supplied functions and core LETKF algorithm. The arrows indicate the order in which the user-supplied functions are executed. They do not imply that one routine calls the other. The observation operators and the global and local domain update are represented by multiple boxes as they are performed by each ensemble member.

vector (*state_p*) from model forecast fields~~into a one-dimensional array, *state_p*. Similarly, *state_p = distribute_state(dim_p, state_p)* distributes analysis (~~. Further, one has to provide a user-supplied function (*state_p = distribute_state(dim_p, state_p)*)

300 such that, after the analysis step, the analysed state vector can be distributed back to model fields for the initialisation of the next forecast cycle. These user-supplied functions allow users to adapt a DA system with different models. For example, as mentioned in Sect. 2.2, optimal state estimation is achieved by ensemble-based Kalman filters under a Gaussian assumption. The state vector collection and distribution function can be used to perform Gaussian anamorphosis where non-Gaussian variables can be transformed to Gaussian variables (Simon and Bertino, 2012).

305 To handle different observations, with the OMI functionality, only three user-supplied functions need to be implemented. ~~One is *dim_obs = init_dim_obs(step, dim_obs_p)*. The primary purpose of the function is to obtain the dimension of observation vector, *dim_obs*, with an initial dimension given by *dim_obs_p* at the current time step, *step*, as implied by its name. In this function, one has to provide further observation information to OMI. The OMI obtains the information in two approaches. One approach is by calling the function:~~ With OMI, users can provide multiple aspects of observation information in the

310 user-supplied function (*dim_obs* = ~~*pyPDAF.PDAF.omi*~~*init_~~gather~~dim_obs(~~i_obs~~step, ~~obs~~dim_p, ivar_obs~~_p, ocoord_p, cradius~~)*. ~~The function returns the total dimension of the~~) that primarily serves the purpose of providing the dimension of observation vector (~~dim_obs) of~~ *dim_obs*) to PDAF. This information for the *i_obs*-th observation type ~~which is returned by the user-supplied function *init_dim_obs*. As function arguments, *pyPDAF.PDAF.omi_gather_obs* provides PDAF with the~~ can be calculated by PDAF function (*dim_obs = pyPDAF.PDAF.omi_gather_obs(i_obs, obs_p, ivar_obs_p, ocoord_p, cradius)*)

315 directly by providing observation vector (~~*ocoord*~~*obs_p*), inverse of the observation error variance (*ivar_obs_p*), the ~~observation coordinates~~ spatial coordinates of observations (*ocoord_p*), and a ~~localisation radius for the current observation type~~ cut-off localisation radius (*cradius*). ~~The other approach sets attributes of the derived data type,~~ In this user-supplied function, one

also sets additional observation attributes (*obs_f* ~~, in PDAF. In *obs_f*, the attributes include the switch of the assimilation of the observation type , the index of the~~) including the switch for assimilating the observation type (*doassim*), the indices of the observations in the state vector ~~, (~~*id_obs_p*), the domain size and the options for distance computation in localisation. ~~While these~~ These attributes can be set by ~~direct initialisation in Fortran, in pyPDAF, these attributes can be set by setter functions ,~~ setter functions (e.g., ~~*id_obs_p* can be set using the pyPDAF function~~ *pyPDAF.PDAF.omi_set_id_obs_p(i_obs, id_obs_p)*).

~~The observation operator is implemented by the~~

Another piece of case-specific information provided by a user-supplied function is the observation operator (*m_state_p = obs_op(step, dim_p, dim_obs_p, state_p, m_state_p)*~~. It takes a~~). The observation operator transforms the state vector (*state_p*) ~~as input and returns a vector in~~ into observation space (*m_state_p*). In ~~our example, it can be handled directly by the OMI function~~ PDAF, observation operators that transform a model field to observations located on grid points (*m_state_p = pyPDAF.PDAF.omi_obs_op_gridpoint(i_obs, state_p, m_state_p)*~~. Note that other observation operators are also available with pyPDAF but~~) are provided. One can construct more complex observation operators in pyPDAF (e.g., Shao and Nerger, 2024) , but is not discussed here ~~. The last user-supplied function related to observations is *dim_obs_l = init_dim_obs_l(domain_p, step, dim_obs, dim_obs_l)* which tells PDAF~~ for the sake of simplicity. In the LETKF, one also has to specify the number of observations being assimilated in ~~the current~~ each local domain (*dim_obs_l*) ~~. This functioncan be simplified by the OMI function~~ in the user-supplied function (*dim_obs_l = init_dim_obs_l(domain_p, step, dim_obs, dim_obs_l)*). In this function, users can simply utilise an OMI function without further implementations (*dim_obs_l = pyPDAF.PDAF.omi_init_dim_obs_l_iso(i_obs, coords_l, locweight, cradius, sradius, dim_obs_l)*~~which automatically handles observation vectors and its~~). The OMI function automatically handles observations and their error variances used in the local domain given the ~~coordinate of~~ coordinates of a local domain (*coords_l*), the type of localisation weight (*locweight*), and the cut-off localisation radius (*cradius*) as well as the support radius of specified localisation function (*sradius*).

~~The domain localisation requires~~ Users must specify information for domain localisation in four additional user-supplied functions. ~~The~~ These include the number of local domains (~~*n_domains_p*) is provided by~~ *n_domains_p = init_n_domains(step, n_domains_p)*), the dimension of the state vector in the *domain_p*-th local domain ~~, *dim_l*, is provided by~~ (*dim_l= init_dim_l(step, domain_p, dim_l)*~~. The~~), the conversion of the full global state vector to a state vector on ~~local domain~~ a local domain (*state_l = g2l_state(step, domain_p, dim_p, state_p, dim_l, state_l)*) and vice versa ~~is controlled by*state_l = g2l_state(step, domain_p, dim_p, state_p, dim_l, state_l)* and~~ (*state_p = l2g_state(step, domain_p, dim_l, state_l, dim_p, state_p)*). The user-supplied function *g2l_state* and *l2g_state* are not used in ~~'PDAFlocal'~~ "PDAFlocal" modules as will be discussed in Sect. 4.2.

~~The pyPDAF analysis step requires two additional~~ In addition to information on the state vector, observations and localisation, the pre- and post-processing of the ensemble is also case-specific. Therefore, this operation must also be performed by a user-supplied ~~functions. The *state_p, uinv, ens_p = prepostprocess(step, dim_p, dim_ens, dim_ens_p, dim_obs_p, state_p, uinv, ens_p, flag)* function is called by PDAF to preprocess the forecast ensemble~~ (function (*state_p, uinv, ens_p = prepostprocess(step, dim_p, dim_ens, dim_ens_p, dim_obs_p, state_p, uinv, ens_p, flag)*)~~before the LETKF and post-process the analysis ensemble~~ . This function allows the user to perform arbitrary operations on the ensemble directly before (pre-processing) or after (~~*ens_p*) after the LETKF assimilated the observations. The~~ post-processing) the analysis step update.

One last piece of case-specific information is the control of DA cycles. In the user-supplied function ~~,~~(*nsteps, doexit, time = next_observation(step, nsteps, doexit, time)*~~, tells PDAF~~), users specify the number of time steps ~~between two DA executions,~~ (*nsteps*) until the next analysis step is computed (thus the duration of the forecast phase). Given the current time step~~and other uninitialised input arguments~~, PDAF also obtains the information of the current model time ~~,~~ (*time*) and a flag for the completion of all DA cycles (*doexit~~in next_observation.~~ To*).

In the completion of the DA system, to control the memory allocation in the DA ~~cycle~~process, the DA system ~~can~~ should be finalised by ~~function~~ a clean-up function (*pyPDAF.PDAF.deallocate()*).

PDAF can handle much more complex cases including non-isotropic localisation, or non-diagonal observation error covariance matrices. Besides LETKF, other filters might require different user-supplied functions as they utilise different case-specific information. The provided pyPDAF example code that exists can support a wide range of filters without changes.

3 ~~Model and DA setup~~Application example

To demonstrate the application of pyPDAF and to evaluate its performance in ~~a coupled DA setup,~~ comparison to PDAF, we set up coupled DA experiments with MAOOAM (De Cruz et al., 2016) version 1.4~~is coupled with PDAF and pyPDAF~~. The original MAOOAM model is ~~implemented~~ written in Fortran that is ~~coupled~~ implemented directly with PDAF, and a wrapper for Python is developed in this study such that MAOOAM can be coupled with pyPDAF. This means that two online DA systems using Fortran and Python ~~respectively are developed~~ are developed respectively to allow for a comparison between the PDAF and pyPDAF implementation. In these DA systems, a suite of twin experiments is carried out using the ensemble transform Kalman filter (ETKF, Bishop et al., 2001) and its domain localisation variant, LETKF.

3.1 ~~Coupled model~~ MAOOAM configuration

The MAOOAM solves a reduced-order non-dimensionalised quasi-geostrophic (QG) equation (De Cruz et al., 2016). Using the beta-plane approximation, the model has a two-layer QG atmosphere component and one-layer QG shallow-water ocean component with both thermal and mechanical coupling. For the atmosphere, the model domain is zonally periodic, and has a no-flux boundary condition meridionally. For the ocean, no-flux boundary conditions are applied in both directions. This setup represents a channel in the atmosphere and a basin in the ocean. The model variables for the two-layer atmosphere are averaged into one layer. Accordingly, MAOOAM consists of four model variables: the atmospheric streamfunction, ψ_a, the atmospheric temperature, T_a, the ocean streamfunction, ψ_o, and the ocean temperature, T_o. The model variables are solved in ~~a~~ spectral space. The spectral basis functions are orthonormal eigenfunctions of the ~~Lapace~~ Laplace operator subject to the boundary condition, and the number of spectral modes is characterised by harmonic wave numbers P, H, M (Cehelsky and Tung, 1987).

Our model configuration adopts the strongly coupled ocean and atmosphere configuration '36st' of Tondeur et al. (2020) using a time step of 0.1 time units corresponding to around 16 minutes. Using the notation of $H^{max}x - P^{max}y$ of De Cruz et al. (2016) with the superscript max the maximum number of harmonic wave numbers, the configuration chooses $2x - 4y$ modes for the

ocean component and $2x-2y$ modes for the atmosphere component. This leads to a total of 36 spectral coefficients with 10 modes for ψ_a and T_a respectively and 8 modes for ψ_o and T_o respectively. The model runs on a rectangular domain with a reference coordinate system of $(x \times y) \in [0, \frac{2\pi}{n}] \times [0, \pi]$, where $n = 1.5$ is the aspect ratio between the x and y dimensions.

In contrast to Tondeur et al. (2020) who assimilate in the spectral space, we assimilate in the physical space in which real observations are usually available. Assimilating in the physical space is not only more realistic but also provides the possibility to investigate the computational efficiency of pyPDAF without changing the model dynamics. This is because the same number of spectral modes can be transformed to different number of grid points. This allows us to focus on the computational cost of the DA. Therefore, for benchmarking computational cost, we conduct a suite of SCDA experiments with $2^k+1 \times 2^k+1$ grid points where $7 \leq k \leq 11$. This gives us state vectors with dimensions ranging from a magnitude of 10^4 to 10^7. We also implement SCDA experiments using LETKF on a grid number size of 257×257 with observations on every 4 and 8 grid points to investigate the efficiency of the domain localisation in pyPDAF.

We integrate MAOOAM with (py)PDAF. As shown in Fig. 1, the key ingredient of coupling MAOOAM with (py)PDAF is the collection and distribution of state ~~vector. In common setups of ocean and atmospheric DA, the observations are available in the physical space. Hence, in the~~ vectors. In the user-supplied function that collects the state vector for pyPDAF (see Fig. 1), spectral modes of the model are transformed from the spectral space to physical space using the transformation equation,

$$f(x,y,t) = \sum_{i=1}^{K} c_i(t) F_i(x,y), \tag{1}$$

where $f(x,y,t)$ is any model variable in the physical space, K is the number of modes, $c_i(t)$ is the spectral coefficient of the model variable, $F_i(x,y)$ is the spectral basis function of mode i outlined in De Cruz et al. (2016). In the user-supplied function that distributes the state vector for pyPDAF (see Fig. 1), the analysis has to be transformed back to the spectral space to initialise the following model forecast. The inverse transformation from the physical space to the spectral space can be obtained by

$$c_i(t) = \frac{n}{2\pi^2} \int_0^{\pi} \int_0^{\frac{2\pi}{n}} f(x,y,t) F_i(x,y) dx dy. \tag{2}$$

where n is the ratio between meridional and zonal extents of the model domain. Here, each basis function corresponds to a spectral coefficient of the model variable. The basis functions are evaluated on an equidistant model grid. The spectral coefficients are obtained via the Romberg numerical integration. ~~To ensure the~~ The accuracy of the numerical integration ~~, the~~ depends on the spatial resolution and the number of grid points ~~is 2^k+1 with $k \in \mathbb{Z}^+$.~~

~~Our model configuration adopts the strongly coupled ocean and atmosphere configuration (36st) of Tondeur et al. (2020) using a time step of 0.1 time units corresponding to around 16 minutes. Using the notation of $H^{max}x - P^{max}y$ of De Cruz et al. (2016) with the superscript max the maximum number of harmonic wave numbers, the configuration chooses $2x-4y$ modes for the ocean component and $2x-2y$ modes for the atmosphere component. This leads to a total of 36 spectral coefficients with 10 modes for ψ_a and T_a respectively and 8 modes for ψ_o and T_o respectively. The model runs on a rectangular domain with a reference coordinate system of $(x \times y) \in [0, \frac{2\pi}{n}] \times [0, \pi]$, where $n = 1.5$ is the aspect ratio between the x and y dimensions.~~

~~In contrast to Tondeur et al. (2020) who assimilate in the spectral space, we assimilate in the physical space in which the observations are usually available. A sensitivity experiment was performed to study the transformation error. The experiment shows that when the~~ with an error of $\mathcal{O}(n^{-2\log_2 n})$ where n is the number of grid points~~reaches~~. Our experiments suggest that the numerical integration error is negligible once we have $(2^7 + 1 \times 2^7 + 1)$ = ~~(129×129), the transformation error becomes negligible and the physical grid pointsresolve the features in the spectral space. In practice, due to the chaotic nature of the model and long simulation time, the error from the transformation can accumulate which subsequently leads to model errors. The transformation between the spectral and physical space allows us to investigate the computational cost of the DA in pyPDAF and PDAF with the same model dynamics. As the ensemble size is determined by the dimension of unstable subspace of the dynamical system, a fixed ensemble size can be used (Tondeur et al., 2020). Therefore, for benchmarking computational cost, we conduct a suite of SCDA experiments with $2^k + 1 \times 2^k + 1$ number of grid points where $7 \leq k \leq 11$. This gives us state vectors with dimension ranging from a magnitude of 10^4 to 10^7. The size of a state vector with around 10^7 elements is closer to operational setups. We also implement SCDA experiments using LETKF on a grid number of 257×257 with observations on every 4 and 8 grid points to investigate the efficiency of the domain localisation in pyPDAF~~(129×129) grid points. In this study, for the sake of efficiency, the transformation between spectral modes and grid points are implemented in Fortran. In pyPDAF systems, the Fortran transformation routines are used by Python with "f2py". This implementation ensures that the numerical computations do not render rounding errors when conducted in different programming languages. Moreover, it also demonstrates that the computationally intensive component of user-supplied functions can be sped up by optimised Fortran code.

3.2 Experiment design

In a twin experiment, a long model run is considered to represent the truth. The model state is simulated with an initial condition sampled in the spectral space ~~which follows~~ following a Gaussian distribution, $\mathcal{N}(0, 0.01)$. The DA experiments are started after ~~10^5~~ 9×10^5 time steps corresponding to around 277 years of model integration to ensure ~~the dynamical consistency of the model state~~that the initial state corrected by the DA follows the trajectory of the dynamical model.

The observations are generated from the truth of the model state based on pre-defined error statistics of the observations. Except for the LETKF experiments, both atmosphere and ocean observations are sampled every 8 model grid points for each model grid setup. In all cases, the observation error standard deviations are set to 50% and 70% of the temporal standard deviation of the true model trajectory at each grid point for the atmosphere and ocean respectively. ~~The resulting standard deviation of the atmosphere observations is on a similar magnitude with the ensemble spread of the free run (cf. Fig. 4) while the magnitude of the observation error in the ocean is typically larger than in the atmosphere in real observing networks. As an example, the obtained standard deviation fields on a grid with 17×17 grid points are shown in Fig. **??**. The observation error standard deviation fields used for generating the synthetic observations. The spatial mean of the error standard deviation is shown in the bracket. With our chosen model configuration, the highest observation error is in the ocean temperature while the ocean streamfunction shows the least uncertainty due to its slow variability.~~ This leads to spatially varying observation errors with regions of larger or smaller observation errors. The atmospheric processes in MAOOAM show variability on shorter timescales

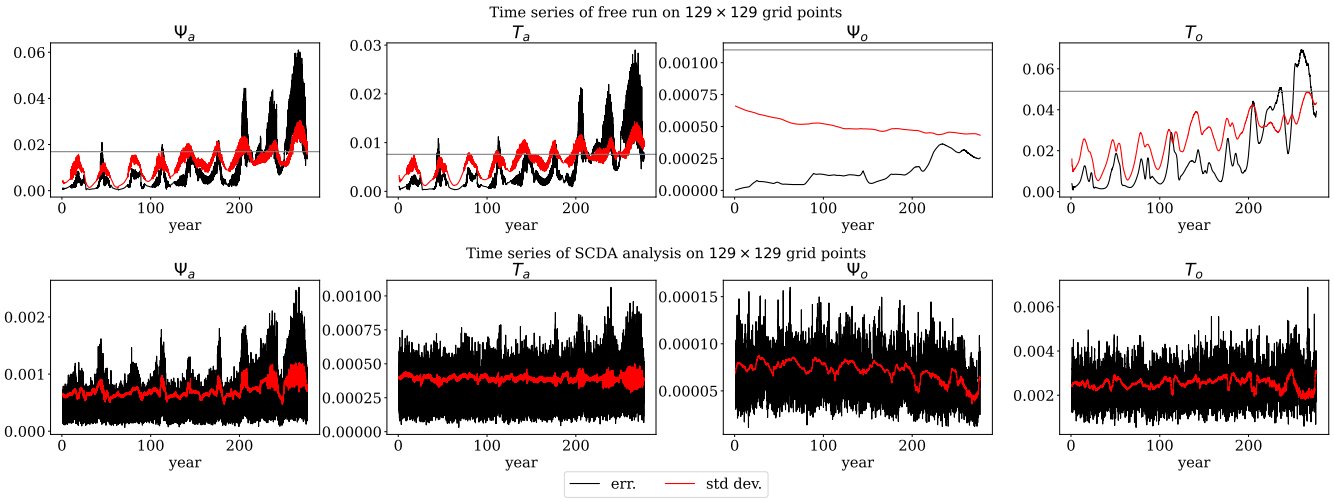

Figure 4. Ensemble ~~spread~~ standard deviation and RMSE of the (top) free run and (bottom) SCDA analysis on a 129×129 grid. Shown are the time series of the spatial mean of ensemble spread (red) and the RMSE of the analysis (black). The atmosphere shows fast variability and oscillatory RMSE of the ensemble mean while the RMSE of the ocean ensemble mean is smooth. The horizontal grey line is the spatial averaged observation error.

than the ocean. Hence, the ocean observations are assimilated around every 7 days (630 time steps) while the atmosphere observations are assimilated around every day (90 time steps). This is in line with the experiment setup in Tondeur et al. (2020).

As shown by Tondeur et al. (2020), DA in the model configuration using 36 spectral coefficients can achieve sufficient accuracy with 15 ensemble members. In this study, 16 ensemble members are used, and each ensemble member runs serially with a single process. Without tuning, a forgetting factor of 0.8 is applied to maintain the ensemble spread~~and ensure a stable DA process.~~. The forgetting factor (Pham et al., 1998) is an efficient approach to multiplicative ensemble inflation in which the covariance matrix is inflated by the inverse of the forgetting factor as shown in the formulation in Nerger et al. (2012).

~~Using the~~ PDAF provides functionality to generate the ensemble. Here, to demonstrate its functionality, we use second-order exact sampling ~~provided by PDAF (see Sect. 2.1),~~ (Pham, 2001), in which the ensemble is generated from a ~~model trajectory by sampling the modelled truth~~ covariance matrix. The covariance matrix is estimated using model states sampled every 10 days over a 100~~years after around~~ -year period, based on the trajectory of the truth model after approximately 1000 years from the ~~beginning~~ start of the simulation. This leads to a covariance matrix with 36 non-zero singular values ~~equaling~~ equalling to the number of spectral modes in the model. The ~~perturbation from the second-order exact sampling~~ ensemble generated from the covariance matrix could violate the dynamical consistency of the model, so that the ensemble would need to be spun up to reach dynamical consistency. To reduce the spin up time, the initial perturbation is scaled down by a factor of 0.2, 0.15, 0.4 for Ψ_a, T_a and T_o respectively. Because the ocean streamfunction has very low variability, its perturbation is unchanged.

The DA experiments are started after 15 days from the beginning of the ensemble generation. The DA experiments are then run for another 9×10^5 time steps which is around 277 years. In this setup, the forecast error is solely a result of inaccuracy of initial conditions. As shown in Fig. 4, the ensemble spread ~~generally~~ captures the trend~~and is in~~ , and has a similar magnitude ~~of~~ as the model forecast error. This suggests that the forecast uncertainty ~~from~~ of the free run ensemble ~~initialised by the second-order exact sampling~~ is able to reflect the forecast errors even though the spread is lower than the RMSE after 200 years.

In the free run (upper panel of Fig. 4), the ocean ~~temperature shows the highest uncertainty of all model variables. The ocean~~ streamfunction shows a very slow error growth rate. This is also shown by the ~~error and ensemble uncertaintywhich are two magnitudes smaller than those of other model variables~~ensemble uncertainty. Sensitivity tests (not shown) suggest that an increased ~~error~~ RMSE of the ocean streamfunction has a ~~significant~~ strong impact on the model dynamics consistent with the theoretical discussion given in Tondeur et al. (2020). The ~~error~~ RMSE of the atmosphere components shows a wave-like behaviour in time. Tondeur et al. (2020) describe the periods associated with fast dynamics with high and oscillatory ~~errors~~ RMSEs as active regimes and the periods associated with slow dynamics with low and stable ~~errors~~ RMSEs as passive regimes.

3.3 Comparison of pyPDAF and PDAF implementation of CDA

As pyPDAF is an interface to PDAF, the same number of user-supplied functions are used for DA systems implemented with pyPDAF and PDAF. As detailed in Sect. 2.4, the ETKF system requires 7 user-supplied functions. For the LETKF system, an additional 5 user-supplied functions are needed. However, as will be discussed in Sect. 4.2, if "PDAFlocal" modules are used, the additional user-supplied function necessary for domain localisation can be reduced to 3.

One of the major advantages of pyPDAF is the ease of implementation. Here, to partially reflect the difference in implementation difficulty between pyPDAF and PDAF, the number of lines of code between pyPDAF and PDAF in each user-supplied functions is compared in Tab. 1. We recognise that such comparison can be inaccurate due to different coding styles and potential unaccounted boilerplate code. Moreover, fewer lines of code do not necessarily represent improved ease of implementation as the DA system setup typically involves scientific research besides code implementation. Nevertheless, we show that Python implementation needs fewer lines of code than Fortran implementation for all user-supplied functions. The reduced implementation difficulty can be attributed to: 1) the Python implementation can make use of efficient third-party Python packages utilising vectorisation avoiding loops and manual implementation; 2) the Python programming language does not require static typing which is required by Fortran; 3) the Python programming language allows for extensible and flexible implementation due to its language features.

4 Results

In this section, ~~we evaluate the DA skill of the~~ to validate the MAOOAM-(py)PDAF online DA system~~using the ETKF~~, we evaluate its DA skill. For the sake of efficiency, the skill of DA is assessed on a domain with 129×129 grid points. To

Table 1. Number of lines of code broken down by user-supplied functions between Fortran and Python implementation of a strongly coupled DA system. The count removes comments and empty blank lines. In the "init_dim_obs" function, we count the total lines of code including functions called within the user-supplied functions and boilerplate code for class definition. The "prepostprocess" is divided into three functions in Python where we count the total lines of code here.

User-supplied functions	Lines of code	
	Fortran	Python
init_ens_pdaf	11	3
collect_state	18	6
distribute_state	41	32
init_dim_obs	261	173
obs_op	25	9
prepostprocess	46	36
init_n_domains	7	3
init_dim_l	9	3
init_dim_obs_l	35	27
g2l_state	11	3
l2g_state	10	3
next_observation	19	11

evaluate the computational efficiency of pyPDAF and PDAF and the potential practical applications of pyPDAF, we compare the wallclock time in the SCDA system.

The online DA systems using PDAF and pyPDAF produce quantitatively the same results in all experiments up to machine precision. This is because the user-supplied functions mainly perform file handling and variable assignments, but no numerical computations. An exception is only the spectral transformation described in Sect. 3.1. To ensure comparable numerical outcome, the numerical computations that affect the forecast and analysis, in particular the spectral transformation, are all conducted in Fortran in this work. These Fortran implementations are used by Python user-supplied functions using "f2py". Note that, when numerical computations involve different programming languages, the model trajectory of the nonlinear system could differ because of errors in the initial conditions arising from rounding errors.

4.1 ~~Effect~~ Skill of ~~coupled~~ data assimilation

As a case study to demonstrate the capability of pyPDAF, both SCDA and WCDA are implemented. In WCDA, ~~the coupling only occurs during the model forecast~~each model component performs DA independently even though the forecast is obtained by the coupled model. This means that the observations only influence their own model component in the analysis step ~~. In this setup, each model component has its own DA system with only two model variables, the streamfunction and temperature, on the same model grid. This~~ which implies two separate DA systems. In an online DA setup in PDAF, two separate state vectors

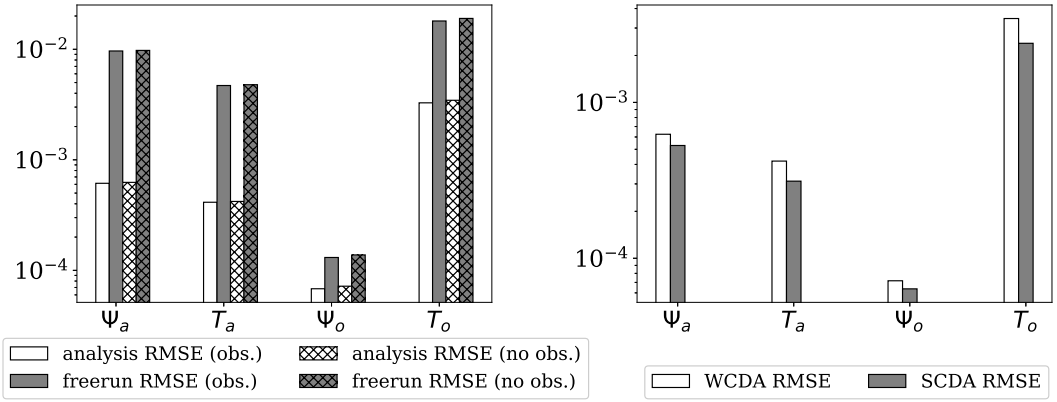

Figure 5. Left: The time-averaged RMSE of the analysis using WCDA and free run where the RMSE of the observed (non-hatched bars), denoted by "obs." in the legend, and unobserved gridpoints (hatched bars), denoted by "no obs.", are compared separately. Right: comparison of RMSEs for weakly and strongly coupled DA for all grid points. The y-axis is plotted in the log-scale.

have to be defined in each analysis step which is not straightforward with PDAF due to its assumption that each analysis step has only one state vector. In the case of AWI-CM in Tang et al. (2021), two separate state vectors were obtained by using ~~a~~ parallelisation, but here the two model components of MAOOAM are run using the same processor. In our implementation we obtain WCDA by resetting the time step counter in PDAF in our implementation such that even if the assimilation of two state vectors are done by using PDAF twice, PDAF only counts it as one analysis time step. An alternative approach could be to use the LETKF method, and define the local state vector as either the atmosphere or ocean variables.

Figure 5 shows that the time averaged RMSE of WCDA is smaller than that of the unconstrained free run. Thus, the error growth is successfully controlled by DA. This also demonstrates that the ETKF leads to a converged analysis even though our observations are less accurate than the forecast at the start of the DA period. The results also show that sparse observations can successfully control errors in regions without observations. This is due to the fact that the model fields are rather smooth leading to long ensemble correlations.

Compared to the WCDA, atmosphere observations influence the ocean part of the state vector and vice versa in the SCDA. This means that the coupling occurs for both the analysis step and model forecast. In this case, the DA system only has one unified state vector that contains the streamfunction and temperature of both model components. The implementation of an online SCDA system aligns with the design of PDAF, and does not require special treatment.

As expected, the SCDA yields lower analysis ~~errors~~ RMSEs than the free run as shown in Fig. 4, and the ~~errors~~ RMSEs are also lower than the WCDA as shown in the right panel of Fig. 5. The improved analysis in the SCDA in each model component is a result of assimilating observations from the other model component. The effective use of these additional observations relies on the error cross-covariance matrix between model components estimated by the forecast ensemble. The improvements suggest a reliable error cross-covariance matrix in the coupled DA system.

~~Time averaged RMSE when only one model component is observed. The y-axis is in log-scale.~~

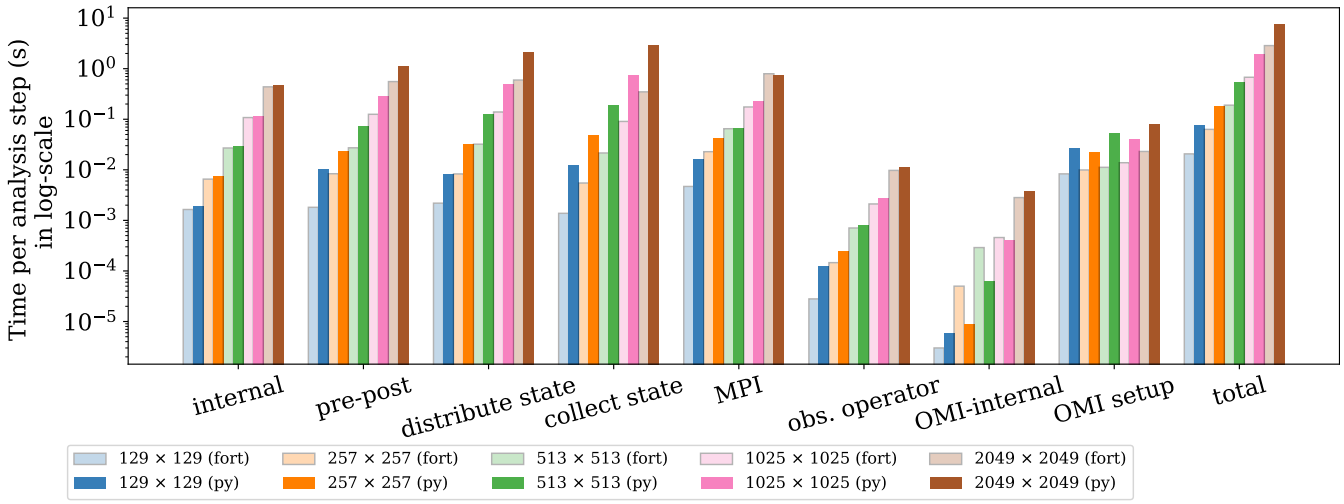

Figure 6. Wall clock time of pyPDAF (dark colour bars) and PDAF (light colour bars) systems per analysis step broken down by functionalities in SCDA ETKF experiments and their total wallclock time per analysis step in ~~log-scale~~**log-scale**.

~~To further show the performance of pyPDAF in a SCDA setup, we carry out experiments in which only one model component is observed. In the SCDA, the analysis increment of a model component without observations relies on the error cross-covariance matrix with the model components that have observations. In this experiment, inflation is only applied to the observed model component to avoid excessive analysis increment to the unobserved model components. The partial inflation is achieved in the post-processing routines as PDAF applies inflation uniformly to the entire state vector by default.~~

~~Figure **??** shows the time-averaged RMSE of fields that are smoothed in time by a moving average as a function of the averaging time-window. The RMSEs of the instantaneous model fields are represented by zero moving average window length. Assimilating observations from the other model component with SCDA can improve the analysis of the unobserved model component. The assimilation not only improves the instantaneous model fields but also the long-term trend of the atmosphere and ocean climate even though the error dynamics of atmosphere and ocean shows strong time-scale differences in Fig. 4. This means that the ocean dynamics benefit from atmosphere observations even if the transient atmosphere processes are smoothed by the moving average. Notably, the RMSE of the ocean streamfunction when only atmosphere observations are assimilated does not decrease monotonically with the moving average window length. This could be explained by the fact that the time averaged ocean streamfunction shows periodic features in time and an moving average of ~ 60 years leads to a time series of nearly constant streamfunction. This improves the skill of the DA . However, this feature is not captured by the analysis that assimilates ocean observations perhaps due to the large observation uncertainties.~~

These results suggest that both pyPDAF and PDAF can be used to implement a DA system as expected.

Table 2. Wall clock time per analysis step of pyPDAF and PDAF for each component of SCDA ETKF using 129×129 grid points and 2049×2049 grid points in seconds. The table also shows the ratio of the wall clock time between Python and Fortran. The wall clock time is the same as the wall clock time shown in Fig. 6.

PDAF component	129×129			2049×2049		
	Python	Fortran	ratio	Python	Fortran	ratio
internal	1.88×10^{-3}	1.64×10^{-3}	1.15	0.47	0.44	1.08
pre-post	1.02×10^{-2}	1.81×10^{-3}	5.62	1.13	0.56	2.04
distribute state	8.37×10^{-3}	2.19×10^{-3}	3.82	2.14	0.60	3.58
collect state	1.23×10^{-2}	1.38×10^{-3}	8.89	2.99	0.35	8.60
MPI	1.62×10^{-2}	4.69×10^{-3}	3.46	0.75	0.79	0.94
obs. operator	1.24×10^{-4}	2.8×10^{-5}	4.43	1.13×10^{-2}	9.73×10^{-3}	1.16
OMI-internal	6×10^{-6}	3×10^{-6}	2.00	3.79×10^{-3}	2.82×10^{-3}	1.34
OMI setup	2.75×10^{-2}	8.31×10^{-3}	3.30	7.98×10^{-2}	2.31×10^{-2}	3.46
total	7.83×10^{-2}	2.07×10^{-2}	3.79	7.66	2.86	2.68

4.2 Computational performance of PDAF and pyPDAF

One motivation of developing a Python interface to PDAF is that the efficient DA algorithms in PDAF can be used by pyPDAF while the user-supplied functions can be developed with ~~the ease of~~ Python. However, the user-supplied functions provided by Python are expected to be slower than a pure Fortran implementation. The slow-down is both a result of lack of compilation in Python and the type cast between Fortran arrays and Python objects. Here we present a comparison of the wall clock time of both PDAF and pyPDAF experiments with standard SCDA broken down to the level of subroutines. Each experiment runs 100 analysis steps, and each experiment is repeated 10 times. The computation runs on the computing facility of University of Reading on a node with two AMD EPYC 7513 32-Core processors which have a 2.6GHz frequency. With 16 ensemble members, each member uses a single processor for model forecast, and the DA is performed serially on a single processor.

The serial DA execution is primarily due to the default parallelisation strategy in PDAF. The ETKF has a straightforward parallelisation since the global transform matrix can be computed in a distributed form followed by a global sum. The LETKF is embarrassingly parallel for each local domain after communicating the necessary observations. Each processor can perform LETKF independently for their local domains. In PDAF, the parallelisation of both ETKF and LETKF is implemented in combination with domain decomposition of the numerical model. In this study, no domain decomposition is carried out for the numerical model itself. Thus, all local domains are located in one single processor for LETKF. The parallelisation strategy of PDAF is further explained in Nerger et al. (2005) and a pyPDAF documentation is available (Parallelisation Strategy, Accessed: 20 March 2

.

As shown in Fig. 6 and Tab. 2, the PDAF-internal procedures (~~labeled~~ labelled 'internal'), which are the core DA algorithm, take nearly the same amount of time per analysis step for PDAF and pyPDAF regardless of the number of grid points. As

expected, the user-supplied functions take more computational time in the DA system based on pyPDAF than PDAF. In this study, the pre- and post-processing of the state vector (~~labeled~~ labelled 'pre-post') calculates the square root of the spatial mean of ensemble variance. The pre- and post-processing is implemented as a user-supplied function (see Sect. 2.4) which is computationally intensive. The intensive computations suit well for the use of the Python JIT compilation. The computational time of the pre- and post-processing increases with the size of the state vector, and Python is in general slower than the Fortran implementation. The difference of wall clock time between the pyPDAF and PDAF-based DA system decreases with increasing state vector size as the overhead in pyPDAF ~~becomes less significant~~ takes smaller portion of the total computation time compared to the floating-point computations. As a comparison ~~, on a 129×129 grid, the PDAF system takes 0.04 seconds while the pyPDAF system takes 0.09 seconds per analysis, thus a factor of 2.15 longer time . However, on a 2049×2049 grid, the PDAF system takes around 40.09 seconds~~ shown in Tab. 2, the ratio of total computational time per analysis step ~~while the pyPDAF system takes 67.96 seconds per analysis step, thus a factor of only 1.7 longer time~~between pyPDAF and PDAF implementation reduces to 2.04 on a 2049×2049 grid from 5.62 on a 129×129 grid. The overhead in the pyPDAF system is also comparatively small in high-dimensional systems for the distribution and collection of state vector (~~labeled~~ labelled 'distribute state' and 'collect state'). For example, the pyPDAF system takes ~~a factor of 2.9 more computational time than~~ 3.82 and 8.89 times of computational time of the PDAF system for 'distribute state' and 'collect state' respectively on a 129×129 grid but ~~only a factor of 1.3 more time is taken by the pyPDAF system than the PDAF system. The overhead in these functions is proportional to the ensemble size as they are called by each ensemble member respectively~~ the ratio is only 2.04 and 3.58 for 'distribute state' and 'collect state' respectively on a 2049×2049 grid. In addition to assigning a state vector to model fields and vice versa in Python, these user-supplied functions perform conversion between physical and spectral space based on Eq. (1) and (2). ~~The~~ As mentioned in Sect. 3.1, the transformation utilises the same Fortran subroutines for both PDAF and pyPDAF system. In the pyPDAF system, the Fortran subroutines are converted to Python functions by ~~'~~"f2py~~'~~". The computational time taken by these functions is proportional to the number of grid points. ~~The~~ In this study, the MPI communications are only used to gather an ensemble matrix from the state vector of each ensemble member located at their specific processor. These communications, which are internal to PDAF ~~which~~ and are not exposed to users, show little differences between pyPDAF and PDAF system.

The wall clock time used for handling observations shows that a pyPDAF DA system is in general slower than a PDAF system. With a low-dimensional state vector, the observation operator (~~labeled~~ labelled 'obs. operator') is slower in a pyPDAF system than PDAF even if the observation operator function only calls a PDAF subroutine provided by OMI. The slow-down of the pyPDAF system is again a result of overhead in the conversion of Fortran and Python arrays. Here, similar to the collection and distribution of the state vector, the function is called by each ensemble member. The overhead ~~becomes less significant~~ takes only a small fraction of the total computation time for high-dimensional state vectors~~when the observation operator computation~~ , while the computing of the observation operator dominates the total computational time of the call. The internal operations of OMI (~~labeled~~ labelled 'OMI internal') are ~~very efficient and~~efficient and, in some cases, the pyPDAF systems can be more efficient than PDAF systems. Our experiments do not show clear benefits between pyPDAF and PDAF system for these operations, as expected especially considering the short wall clock time at an order of 10^{-6} with 129×129 grid points

used in these operations. The setup of the OMI functionality is implemented in the user-supplied function of *init_dim_obs* (see Sect. 2.4). This includes reading and processing the observation data and their errors. In this case, the pyPDAF-based system is more expensive than the PDAF system. The pyPDAF system ~~is 2.15 (8.57) times slower in executing~~ takes 3.30 (3.46) times of the computational time used to execute *init_dim_obs* ~~than the~~ in PDAF system on a 129×129 (2049×2049) grid. The relative increase ~~is due to~~ of computational time between the pyPDAF and PDAF system is not evident even though a larger number of observations ~~that~~ needs to be processed.

Our comparison shows that the interfacing between Python and Fortran yields overheads in the pyPDAF system even if we utilise JIT compilation of Python. Users need to consider a trade-off between these overheads and the ease of implementation in pyPDAF compared to PDAF. The differences of the computational cost can ~~be less significant~~ take a smaller portion of the total computation time for high-dimensional systems for the ETKF DA system without localisation due to increased numerical computations.

In practice, localisation is used to avoid sampling errors in high-dimensional weather and climate systems. To make full use of the computational resources, these high-dimensional systems are parallelised by domain decomposition. PDAF exploits the feature of these models for domain localisation where the state vector is also domain decomposed. Here, we choose a domain with 257×257 grid points to assess the LETKF with a cut-off localisation radius of 1 non-dimensionalised spatial unit. This corresponds to 3000 km covering around a third of the domain. As no domain decomposition is implemented for MAOOAM, each processor contains ~~$257 \times 257 \times 4$~~ 257×257 local domains which is similar to the number of local domains used in a single processor of a domain decomposed global climate model.

For each local domain, the LETKF computes an analysis using observations with a localisation cut-off radius. Hence, the computational cost depends on the observation density. To investigate the effect of increased intensity of computations on the pyPDAF overhead, we add experiments that observe every 4 grid points.

As shown in Fig. 7, the increased observation density leads to an increase in computational time for the internal operations, observation operator, and the OMI-internal operations due to the larger number of locally assimilated observations. For example, for applying the observation operator, when every 8 grid points are observed, $\sim 2.4 \times 10^{-4}$ s and $\sim 1.4 \times 10^{-4}$ s are respectively used in pyPDAF and PDAF systems, but in the case of observations for every 4 grid points, $\sim 8.4 \times 10^{-4}$ s and $\sim 7.4 \times 10^{-4}$ s is used in pyPDAF and PDAF systems respectively. The increased observation density shows little influence on the computational cost of other user-supplied functions. For example, in the case of the OMI setup, when every 8 grid points are observed, ~ 0.047 s and ~ 0.014 s is used in the pyPDAF and PDAF systems respectively, and in the case of observations for every 4 grid points, a similar computational time of ~ 0.035 s and ~ 0.0137 s is used in pyPDAF and PDAF systems respectively. However, as the increased observation density leads to more intensive computations, this mitigates the gap of the total computational time between pyPDAF and PDAF system. In particular, as shown in Tab. 3 the run times for the internal operations of PDAF ~~(not shown)~~ and OMI ('OMI-internal') dominate the overall run time of the analysis step and show little difference for the pyPDAF and PDAF DA systems. Here, when executing 'OMI-internal' operations, when every 8 grid points are observed, ~ 3.07 s and ~ 2.69 s is used in pyPDAF and PDAF systems respectively, but in the case of observations for every 4 grid points, ~ 11.04 s and ~ 10.74 s is used in pyPDAF and PDAF systems respectively.

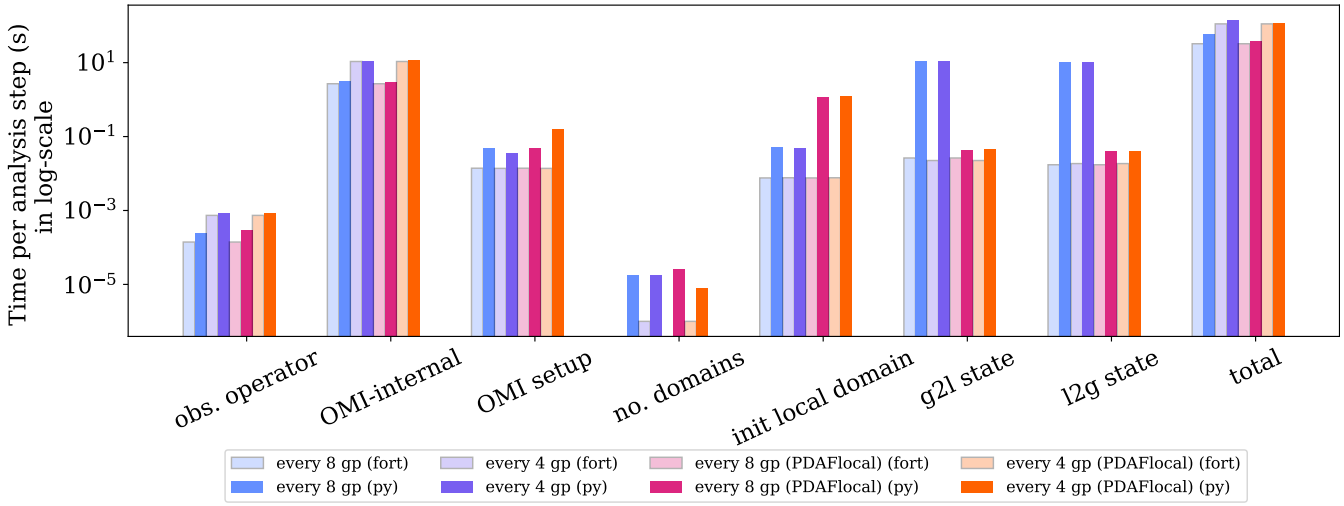

Figure 7. Wall clock time of pyPDAF (light colour bars) and PDAF (dark colour bars) system per analysis step broken down by functionalities in SCDA LETKF experiments and their total wallclock time per analysis step in ~~log-scale~~**log-scale** with a 257×257 grid points. The left four bars (blue and purple bars) represent the case without using the PDAFlocal module while the rest uses the PDAFlocal module. For the sake of conciseness, the functionalities shared by both ETKF and LETKF are omitted. The computational time of PDAF system for 'no. domains' is negligible when every 8 grid points are observed which lead to an empty bar.

Table 3. Wall clock time per analysis step of pyPDAF and PDAF for each component of SCDA LETKF using 257×257 grid points in seconds where observations are taken every 4 grid points. The table also shows the ratio of the wall clock time between Python and Fortran. The wall clock time is the same as the wall clock time shown in Fig. 7.

PDAF component	Python	Fortran	ratio	Python (PDAFlocal)	Fortran (PDAFlocal)	ratio
internal	18.33	17.85	1.03	18.24	17.85	1.02
pre-post	2.35×10^{-2}	9.01×10^{-3}	2.61	2.36×10^{-2}	9.01×10^{-3}	2.62
distribute state	3.16×10^{-2}	8.28×10^{-3}	3.81	3.16×10^{-2}	8.28×10^{-3}	3.81
collect state	4.68×10^{-2}	5.55×10^{-3}	8.44	4.67×10^{-2}	5.55×10^{-3}	8.42
MPI	3.13×10^{-2}	2.59×10^{-2}	1.21	5.27×10^{-2}	2.59×10^{-2}	2.03
obs. operator	8.40×10^{-4}	7.37×10^{-4}	1.14	8.33×10^{-4}	7.37×10^{-4}	1.13
OMI-internal	11.04	10.74	1.03	11.25	10.74	1.05
OMI setup	3.53×10^{-2}	1.38×10^{-2}	2.56	0.16	1.38×10^{-2}	11.59
no. domains	1.80×10^{-5}	1.00×10^{-6}	18.00	8.0×10^{-6}	1.0×10^{-6}	8.00
init local domain	4.96×10^{-2}	7.68×10^{-3}	6.45	1.26	7.68×10^{-3}	164.68
g2l state	10.69	2.25×10^{-2}	475.52	4.58×10^{-2}	2.25×10^{-2}	2.04
l2g state	10.36	1.86×10^{-2}	557.48	3.99×10^{-2}	1.86×10^{-2}	2.15
total	140.82	111.88	1.26	118.95	111.88	1.06

We notice ~~significant~~ large overhead in the pyPDAF system for user-supplied functions related to domain localisation. The ~~increased computational time when the number of domains is specified (labeled~~ 'no. domains' ~~) is still of an order of 10^{-4}~~ user-supplied function takes $\sim 1.8 \times 10^{-5}$ s per analysis step ~~which is negligible. The computation is 5.65 times slower in pyPDAF than the PDAF~~ for pyPDAF system but only $\sim 1. \times 10^{-6}$ s is taken by the PDAF system. The latter can be negligible when every 8 grid points are observed. In this user-supplied function, only one assignment is executed in the user-supplied function. Therefore, the overhead is primarily a result of conversion between the interoperation between Fortran and Python. This operation has little impact on the overall efficiency of the system. The computation takes 6.45 times of computational time of PDAF system in the pyPDAF system for the function specifying the dimension of the local state vector ('init local domain') ~~.~~ ~~as shown in Tab. 3.~~ The increased computational cost is a result of repeated execution of the user-supplied functions for each local domain. Specifically, in our experiment, this user-supplied function is used $257 \times 257 \times 4$ times per analysis step. The overhead is even higher for the user-supplied functions that convert between local state vector and global state vector ('g2l state' and 'l2g state' respectively), which are called for each ensemble member, due to the conversion of arrays instead of integers. In this experiment, the execution of these routines in pyPDAF ~~system is around 400~~ can be even more than 500 times slower than in the PDAF system. As these operations are not computationally intensive, the overhead cannot be mitigated by JIT compilation. Without modifications in the PDAF workflow, the overhead can become comparatively ~~less significant~~ smaller with high observation density arising from increased computational cost of other routines, or increased parallelisation of model domains leading to reduced number of local domains on each processor.

To overcome ~~this~~ the specific run time issue of 'g2l state' and 'l2g state', we developed a $PDAFlocal$ module in PDAF, included in release version 2.3, where the user-supplied functions of 'g2l state' and 'l2g state' are circumvented in the PDAF interface as their operations are performed in the compiled Fortran code of $PDAFlocal$. This leads to similar computational cost of these functions ~~between~~ in the pyPDAF and PDAF ~~system~~systems. With $PDAFlocal$, users need to implement an index vector providing the relationship between the state vector in the current local domain and the global state vector when local domain is initialised in 'init local domain'. Due to this, with PDAFlocal, we see an increased computational time in 'init local domain' in pyPDAF, which is around ~~150~~ 160 times slower than the PDAF system. ~~The~~ However, this pyPDAF overhead for 'init local domain' is smaller than that of 'g2l state' and 'l2g state' ~~(around 400 times slowdown) due to reduced~~ due to the different types of operations and hence a lower number of array conversions between Fortran and Python in 'init local domain'. Further, only one instead of three user-supplied functions are implemented in Python. ~~Due to this~~With the enhancement by $PDAFlocal$, the total computing time is nearly equal for pyPDAF and PDAF with only $6\% - 13\%$ higher time for pyPDAF.

As both numerical computation and user-supplied functions can be sensitive to the number of ensemble members, we further compare the computational time for different ensemble members with 257×257 grid points observed by every 8 grid points. As shown in Tab. 4, the ETKF takes longer computational time for a larger ensemble. Consistent with other experiments and as expected, the internal PDAF operations take similar computational time between Fortran-based PDAF and Python-based pyPDAF. For all user-supplied functions, compared to the pure Fortran implementation, the pyPDAF leads to increased computational time. The overhead depends on the specific implementation of each function. For example, in the state vector collection and distribution, even though the computational cost should be theoretically proportional to the ensemble size,

Table 4. Wall clock time per analysis step of pyPDAF and PDAF for each component of SCDA ETKF in seconds with different ensemble members using 257×257 grid points where observations are taken every 8 grid points. The table also shows the ratio of the wall clock time between Python and Fortran.

PDAF component	64 members			128 members		
	Python	Fortran	ratio	Python	Fortran	ratio
internal	0.07	0.07	1.03	0.27	0.26	1.03
pre-post	0.15	0.02	6.52	0.25	0.04	6.41
distribute state	0.04	0.01	3.88	0.04	0.01	3.98
collect state	0.06	0.01	8.89	0.05	0.01	9.08
MPI	0.14	0.08	1.69	0.44	0.52	0.84
obs. operator	1.2×10^{-3}	7.1×10^{-4}	1.67	2.6×10^{-3}	1.4×10^{-3}	1.83
OMI-internal	8.1×10^{-5}	1.1×10^{-4}	0.72	1.7×10^{-3}	2.8×10^{-4}	5.97
OMI setup	0.04	0.01	3.46	0.04	0.01	3.78
total	0.50	0.21	2.40	1.10	0.86	1.28

the overall overhead is stable regardless of the ensemble size. In the pre- and post-processing functions, the overhead relative to the Fortran implementation gets smaller with increased ensemble size as the numerical computations take a higher proportion of the total computational time. The effect of increased ensemble size is also revealed in the observation-related functions. In both the user-supplied function for the observation operator and the internal PDAFomi functions, the computational time increases with the ensemble size. In our specific experiment setup, the overhead does not show large differences with different ensemble size for these observation-related functions. Nevertheless, when the overall computational time between the Fortran and pyPDAF implementation is compared, increasing the ensemble size leads to comparatively lower overhead due to increased numerical computations.

We recognise that the exact computational time can be case-specific. For example, we can postulate that, compared to this study, the overhead can be comparatively smaller for computational intensive user-supplied functions where JIT can be used. This could be the case when correlated observation error covariances are used. Even though this study only investigates the commonly used ETKF and LETKF, the relative run times of pure PDAF and pyPDAF should be similar for other global and local filters. This expectation results from the algorithmic similarity of many filters and the fact that the user routines which are coded in Python when using pyPDAF are mainly the same. However, the overhead may also vary depending on the DA algorithms, in particular for variants of 3DVar. These results demonstrate that pyPDAF ~~can~~ has the potential to be used with high-dimensional systems with ~~slightly~~ some increased overhead per analysis step.

5 Conclusions

We introduce the Python package pyPDAF, which provides an interface to the Parallel Data Assimilation Framework (PDAF). We outline its implementation and design. pyPDAF allows for a Python-based DA system for models coded in Python or interfaced to Python. Furthermore it allows for the implementation of a Python-based offline DA system where the DA is performed separately from the model and data is exchanged between the model and DA code through files. The pyPDAF package allows one to implement user-supplied functions in Python for flexible code development while the DA system still benefits from PDAF's efficient DA algorithm implementation in Fortran.

Using a CDA setup, we demonstrate that pyPDAF can be used with the Python model MAOOAM. Both strongly coupled data assimilation (SCDA) and weakly coupled data assimilation (WCDA) are demonstrated. Our results confirm that the SCDA performs better than WCDA~~, and additional observations from other model components can improve the overall performance of DA using SCDA. We also investigate the scenario where only one model component is observed. In this case, the error cross-covariance matrix from the ETKF is sufficiently reliable for updating the unobserved model variables leading to improved analyses states for both observed and un-observed model variables. We also show that the DA can improve the long-term trend of the model state in the MAOOAM model~~. The advantage of pyPDAF in terms of the ease of implementation is reflected by a comparison of the number of lines of code by user-supplied functions in the SCDA setup. The pyPDAF implementation consistently uses fewer lines of code showcasing the requirement for a lower implementation effort than PDAF implementation.

Using the SCDA setup, the computational costs of using pyPDAF and a Fortran-only implementation with PDAF are compared. We show that the computational time stays ~~the same~~ similar for the core DA algorithm executed in PDAF while pyPDAF yields an overhead in user-supplied functions. This overhead is a result of both the Python implementation and the requirement of data conversion between Python and Fortran. These overheads become comparatively ~~less significant~~ smaller when the analysis becomes computationally more intensive with increased spatial resolution or observation density. To mitigate the overhead in domain localisation implementations, we ~~introduce a new~~ introduced a new module "PDAFlocal" ~~module~~ in PDAF such that a DA system using pyPDAF can achieve similar computational cost as a pure Fortran based system. This module is included ~~in~~ since the release v2.3 of PDAF and is now also recommended for the Fortran implementation due to the lower implementation effort. We note that JIT compilation or 'f2py' can be used with the Python user-supplied functions for computationally intensive tasks to speed up the Python DA system. ~~Our~~ In the scope of our specific experiment setup, our benchmark shows that ~~, with a global filter,~~ 70% more time is used ~~, and with a domain localised filter ,~~ with the global filter while only $6\% - 13\%$ more time is ~~used~~ required with a domain-localized filter when applying the Python DA system build with pyPDAF in a high-dimensional dynamical ~~systems~~system.

We recognise that the computational cost of the pyPDAF and PDAF can vary case-by-case. Our results demonstrate that the additional "PDAFlocal" module was essential to mitigate the computational overhead in the case of domain localisation. When pyPDAF is used for other DA algorithms and applications, potential efficiency gain can be implemented in future releases of both PDAF and pyPDAF as both pyPDAF and PDAF are still under active development and maintenance.

pyPDAF opens the possibility to apply sophisticated efficient parallel ensemble DA to large-scale Python models such as machine learning models. pyPDAF also allows for the construction of efficient offline Python DA systems. In particular, pyPDAF can be integrated to machine learning models as long as the ~~state vector~~ data structures of such models can be converted to ~~numpy arrays~~the numpy arrays used by pyPDAF. A pyPDAF-based DA system allows users to utilise sophisticated parallel ensemble DA methods. However, a pyPDAF system does not support GPU parallelisation like TorchDA (Cheng et al., 2025), which is designed based on the machine learning framework pyTorch. The TorchDA package may also have ~~limitation on~~ limitations for the application of DA on machine learning models implemented by other frameworks.

Code availability. The Fortran and Python code and corresponding configuration and plotting scripts including the randomly generated initial condition for the coupled DA experiments are available at: https://doi.org/10.5281/zenodo.11367123. The MAOOAM V1.4 model used for our experiments is available at https://github.com/Climdyn/MAOOAM/releases/tag/v1.4 with a version available at https://doi.org/10.5281/zenodo.1308192. The Fortran version of the experiment depends on PDAF V2.3 which is released at https://doi.org/10.5281/zenodo.13789628 and can be also found at https://github.com/PDAF/PDAF/releases/tag/PDAF_V2.3 (Nerger, 2024). The source code of pyPDAF is available at https://github.com/yumengch/pyPDAF/releases/tag/v1.0.0 with the exactly same version at https://doi.org/10.5281/zenodo.10950130.

Author contributions. YC coded and distributed the pyPDAF package, conducted the experiments, performed the data analysis, and wrote the paper. LN coded PDAF and the PDAFlocal module. All authors contribute to the conceptual experiment design and the paper writing.

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

Acknowledgements. The authors acknowledge the UK National Environment Research Council's support for the National Centre for Earth Observation (Contract Number: PR140015).

References

Abernathey, R., rochanotes, Ross, A., Jansen, M., Li, Z., Poulin, F. J., Constantinou, N. C., Sinha, A., Balwada, D., SalahKouhen, Jones, S., Rocha, C. B., Wolfe, C. L. P., Meng, C., van Kemenade, H., Bourbeau, J., Penn, J., Busecke, J., Bueti, M., and Tobias: pyqg/pyqg: v0.7.2, Zenodo [code], https://doi.org/10.5281/zenodo.6563667, 2022.

Ahmed, S. E., Pawar, S., and San, O.: PyDA: A Hands-On Introduction to Dynamical Data Assimilation with Python, Fluids, 5, https://doi.org/10.3390/fluids5040225, 2020.

Anderson, J., Hoar, T., Raeder, K., Liu, H., Collins, N., Torn, R., and Avellano, A.: The Data Assimilation Research Testbed: A Community Facility, Bulletin of the American Meteorological Society, 90, 1283 – 1296, https://doi.org/https://doi.org/10.1175/2009BAMS2618.1, 2009.

Anderson, J. L.: An Ensemble Adjustment Kalman Filter for Data Assimilation, Monthly Weather Review, 129, 2884 – 2903, https://doi.org/10.1175/1520-0493(2001)129<2884:AEAKFF>2.0.CO;2, 2001.

Bannister, R. N.: A review of operational methods of variational and ensemble-variational data assimilation, Quarterly Journal of the Royal Meteorological Society, 143, 607–633, https://doi.org/https://doi.org/10.1002/qj.2982, 2017.

Bi, K., Xie, L., Zhang, H., Chen, X., Gu, X., and Tian, Q.: Accurate medium-range global weather forecasting with 3D neural networks, Nature, 619, 533–538, https://doi.org/10.1038/s41586-023-06185-3, 2023.

Bishop, C. H., Etherton, B. J., and Majumdar, S. J.: Adaptive Sampling with the Ensemble Transform Kalman Filter. Part I: Theoretical Aspects, Monthly Weather Review, 129, 420 – 436, https://doi.org/https://doi.org/10.1175/1520-0493(2001)129<0420:ASWTET>2.0.CO;2, 2001.

Bonavita, M., Hólm, E., Isaksen, L., and Fisher, M.: The evolution of the ECMWF hybrid data assimilation system, Quarterly Journal of the Royal Meteorological Society, 142, 287–303, https://doi.org/https://doi.org/10.1002/qj.2652, 2016.

Bruggeman, J., Bolding, K., Nerger, L., Teruzzi, A., Spada, S., Skákala, J., and Ciavatta, S.: EAT v1.0.0: a 1D test bed for physical–biogeochemical data assimilation in natural waters, Geoscientific Model Development, 17, 5619–5639, https://doi.org/10.5194/gmd-17-5619-2024, 2024.

Caron, J.-F., Milewski, T., Buehner, M., Fillion, L., Reszka, M., Macpherson, S., and St-James, J.: Implementation of Deterministic Weather Forecasting Systems Based on Ensemble–Variational Data Assimilation at Environment Canada. Part II: The Regional System, Monthly Weather Review, 143, 2560 – 2580, https://doi.org/10.1175/MWR-D-14-00353.1, 2015.

Carrassi, A., Bocquet, M., Bertino, L., and Evensen, G.: Data assimilation in the geosciences: An overview of methods, issues, and perspectives, WIREs Climate Change, 9, e535, https://doi.org/https://doi.org/10.1002/wcc.535, 2018.

Cehelsky, P. and Tung, K. K.: Theories of Multiple Equilibria and Weather Regimes—A Critical Reexamination. Part II: Baroclinic Two-Layer Models, Journal of Atmospheric Sciences, 44, 3282 – 3303, https://doi.org/10.1175/1520-0469(1987)044<3282:TOMEAW>2.0.CO;2, 1987.

Cheng, S., Min, J., Liu, C., and Arcucci, R.: TorchDA: A Python package for performing data assimilation with deep learning forward and transformation functions, Computer Physics Communications, 306, 109 359, https://doi.org/https://doi.org/10.1016/j.cpc.2024.109359, 2025.

Clayton, A. M., Lorenc, A. C., and Barker, D. M.: Operational implementation of a hybrid ensemble/4D-Var global data assimilation system at the Met Office, Quarterly Journal of the Royal Meteorological Society, 139, 1445–1461, https://doi.org/https://doi.org/10.1002/qj.2054, 2013.

De Cruz, L., Demaeyer, J., and Vannitsem, S.: The Modular Arbitrary-Order Ocean-Atmosphere Model: MAOOAM v1.0, Geoscientific Model Development, 9, 2793–2808, https://doi.org/10.5194/gmd-9-2793-2016, 2016.

de Rosnay, P., Browne, P., de Boisséson, E., Fairbairn, D., Hirahara, Y., Ochi, K., Schepers, D., Weston, P., Zuo, H., Alonso-Balmaseda, M., Balsamo, G., Bonavita, M., Borman, N., Brown, A., Chrust, M., Dahoui, M., Chiara, G., English, S., Geer, A., Healy, S., Hersbach, H., Laloyaux, P., Magnusson, L., Massart, S., McNally, A., Pappenberger, F., and Rabier, F.: Coupled data assimilation at ECMWF: current status, challenges and future developments, Quarterly Journal of the Royal Meteorological Society, 148, 2672–2702, https://doi.org/https://doi.org/10.1002/qj.4330, 2022.

Döll, P., Hasan, H. M. M., Schulze, K., Gerdener, H., Börger, L., Shadkam, S., Ackermann, S., Hosseini-Moghari, S.-M., Müller Schmied, H., Güntner, A., and Kusche, J.: Leveraging multi-variable observations to reduce and quantify the output uncertainty of a global hydrological model: evaluation of three ensemble-based approaches for the Mississippi River basin, Hydrology and Earth System Sciences, 28, 2259–2295, https://doi.org/10.5194/hess-28-2259-2024, 2024.

Evensen, G.: Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics, Journal of Geophysical Research: Oceans, 99, 10 143–10 162, https://doi.org/https://doi.org/10.1029/94JC00572, 1994.

Evensen, G., Vossepoel, F. C., and van Leeuwen, P. J.: Data assimilation fundamentals: A unified formulation of the state and parameter estimation problem, Springer Nature, 2022.

Eyring, V., Bony, S., Meehl, G. A., Senior, C. A., Stevens, B., Stouffer, R. J., and Taylor, K. E.: Overview of the Coupled Model Intercomparison Project Phase 6 (CMIP6) experimental design and organization, Geoscientific Model Development, 9, 1937–1958, https://doi.org/10.5194/gmd-9-1937-2016, 2016.

Feng, L., Palmer, P., Bösch, H., and Dance, S.: Estimating surface CO 2 fluxes from space-borne CO 2 dry air mole fraction observations using an ensemble Kalman Filter, Atmospheric chemistry and physics, 9, 2619–2633, 2009.

filterpy PyPI: https://pypi.org/project/filterpy/, last access: 2024-08-29.

Hamill, T. M., Whitaker, J. S., and Snyder, C.: Distance-Dependent Filtering of Background Error Covariance Estimates in an Ensemble Kalman Filter, Monthly Weather Review, 129, 2776 – 2790, https://doi.org/10.1175/1520-0493(2001)129<2776:DDFOBE>2.0.CO;2, 2001.

Hamill, T. M., Whitaker, J. S., Fiorino, M., and Benjamin, S. G.: Global Ensemble Predictions of 2009's Tropical Cyclones Initialized with an Ensemble Kalman Filter, Monthly Weather Review, 139, 668 – 688, https://doi.org/10.1175/2010MWR3456.1, 2011.

Hersbach, H., Bell, B., Berrisford, P., Hirahara, S., Horányi, A., Muñoz-Sabater, J., Nicolas, J., Peubey, C., Radu, R., Schepers, D., Simmons, A., Soci, C., Abdalla, S., Abellan, X., Balsamo, G., Bechtold, P., Biavati, G., Bidlot, J., Bonavita, M., De Chiara, G., Dahlgren, P., Dee, D., Diamantakis, M., Dragani, R., Flemming, J., Forbes, R., Fuentes, M., Geer, A., Haimberger, L., Healy, S., Hogan, R. J., Hólm, E., Janisková, M., Keeley, S., Laloyaux, P., Lopez, P., Lupu, C., Radnoti, G., de Rosnay, P., Rozum, I., Vamborg, F., Villaume, S., and Thépaut, J.-N.: The ERA5 global reanalysis, Quarterly Journal of the Royal Meteorological Society, 146, 1999–2049, https://doi.org/https://doi.org/10.1002/qj.3803, 2020.

Houtekamer, P. L., Mitchell, H. L., Pellerin, G., Buehner, M., Charron, M., Spacek, L., and Hansen, B.: Atmospheric Data Assimilation with an Ensemble Kalman Filter: Results with Real Observations, Monthly Weather Review, 133, 604 – 620, https://doi.org/10.1175/MWR-2864.1, 2005.

Hu, C.-C. and van Leeuwen, P. J.: A particle flow filter for high-dimensional system applications, Quarterly Journal of the Royal Meteorological Society, 147, 2352–2374, https://doi.org/https://doi.org/10.1002/qj.4028, 2021.

Hunt, B. R., Kostelich, E. J., and Szunyogh, I.: Efficient data assimilation for spatiotemporal chaos: A local ensemble transform Kalman filter, Physica D: Nonlinear Phenomena, 230, 112–126, https://doi.org/https://doi.org/10.1016/j.physd.2006.11.008, 2007.

Installation - pyPDAF documentation: https://yumengch.github.io/pyPDAF/, last access: 2025-03-25.

Kalnay, E., Sluka, T., Yoshida, T., Da, C., and Mote, S.: Review article: Towards strongly coupled ensemble data assimilation with additional improvements from machine learning, Nonlinear Processes in Geophysics, 30, 217–236, https://doi.org/10.5194/npg-30-217-2023, 2023.

Kurth, T., Subramanian, S., Harrington, P., Pathak, J., Mardani, M., Hall, D., Miele, A., Kashinath, K., and Anandkumar, A.: FourCastNet: Accelerating Global High-Resolution Weather Forecasting Using Adaptive Fourier Neural Operators, in: The Platform for Advanced Scientific Computing 2023, Association for Computing Machinery, New York, NY, USA, https://doi.org/10.1145/3592979.3593412, 2023.

Kurtz, W., He, G., Kollet, S. J., Maxwell, R. M., Vereecken, H., and Hendricks Franssen, H.-J.: TerrSysMP–PDAF (version 1.0): a modular high-performance data assimilation framework for an integrated land surface–subsurface model, Geoscientific Model Development, 9, 1341–1360, https://doi.org/10.5194/gmd-9-1341-2016, 2016.

Lam, R., Sanchez-Gonzalez, A., Willson, M., Wirnsberger, P., Fortunato, M., Alet, F., Ravuri, S., Ewalds, T., Eaton-Rosen, Z., Hu, W., Merose, A., Hoyer, S., Holland, G., Vinyals, O., Stott, J., Pritzel, A., Mohamed, S., and Battaglia, P.: Learning skillful medium-range global weather forecasting, Science, 382, 1416–1421, https://doi.org/10.1126/science.adi2336, 2023.

Losa, S. N., Danilov, S., Schröter, J., Nerger, L., Maβmann, S., and Janssen, F.: Assimilating NOAA SST data into the BSH operational circulation model for the North and Baltic Seas: Inference about the data, Journal of Marine Systems, 105-108, 152–162, https://doi.org/https://doi.org/10.1016/j.jmarsys.2012.07.008, 2012.

McGibbon, J., Brenowitz, N. D., Cheeseman, M., Clark, S. K., Dahm, J. P. S., Davis, E. C., Elbert, O. D., George, R. C., Harris, L. M., Henn, B., Kwa, A., Perkins, W. A., Watt-Meyer, O., Wicky, T. F., Bretherton, C. S., and Fuhrer, O.: fv3gfs-wrapper: a Python wrapper of the FV3GFS atmospheric model, Geoscientific Model Development, 14, 4401–4409, https://doi.org/10.5194/gmd-14-4401-2021, 2021.

Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.1, https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf, 2023.

Nerger, L.: Data assimilation for nonlinear systems with a hybrid nonlinear Kalman ensemble transform filter, Quarterly Journal of the Royal Meteorological Society, 148, 620–640, https://doi.org/https://doi.org/10.1002/qj.4221, 2022.

Nerger, L.: PDAF Version 2.3, Zenodo [code], https://doi.org/10.5281/zenodo.13789628, 2024.

Nerger, L. and Hiller, W.: Software for ensemble-based data assimilation systems—Implementation strategies and scalability, Computers & Geosciences, 55, 110–118, https://doi.org/https://doi.org/10.1016/j.cageo.2012.03.026, ensemble Kalman filter for data assimilation, 2013a.

Nerger, L. and Hiller, W.: Software for Ensemble-based Data Assimilation Systems - Implementation Strategies and Scalability, Computers & Geosciences, 55, 110–118, 2013b.

Nerger, L., Hiller, W., and Schröter, J.: PDAF - The parallel data assimilation framework: experiences with Kalman filtering, in: Use of High Performance Computing in Meteorology, pp. 63–83, https://doi.org/10.1142/9789812701831_0006, 2005.

Nerger, L., Janjić, T., Schröter, J., and Hiller, W.: A Unification of Ensemble Square Root Kalman Filters, Monthly Weather Review, 140, 2335 – 2345, https://doi.org/https://doi.org/10.1175/MWR-D-11-00102.1, 2012.

Nerger, L., Tang, Q., and Mu, L.: Efficient ensemble data assimilation for coupled models with the Parallel Data Assimilation Framework: example of AWI-CM (AWI-CM-PDAF 1.0), Geoscientific Model Development, 13, 4305–4321, https://doi.org/10.5194/gmd-13-4305-2020, 2020.

Parallelisation Strategy: https://yumengch.github.io/pyPDAF/parallel.html, Accessed: 20 March 2025.

PDAF - the Parallel Data Assimilation Framework: https://pdaf.awi.de/, last access: 2024-02-13.

Penny, S. G. and Hamill, T. M.: Coupled data assimilation for integrated earth system analysis and prediction, Bulletin of the American Meteorological Society, 98, ES169–ES172, 2017.

Pham, D. T.: Stochastic Methods for Sequential Data Assimilation in Strongly Nonlinear Systems, Monthly Weather Review, 129, 1194 – 1207, https://doi.org/https://doi.org/10.1175/1520-0493(2001)129<1194:SMFSDA>2.0.CO;2, 2001.

Pham, D. T., Verron, J., and Christine Roubaud, M.: A singular evolutive extended Kalman filter for data assimilation in oceanography, Journal of Marine Systems, 16, 323–340, https://doi.org/https://doi.org/10.1016/S0924-7963(97)00109-7, 1998.

Pohlmann, H., Brune, S., Fröhlich, K., Jungclaus, J. H., Sgoff, C., and Baehr, J.: Impact of ocean data assimilation on climate predictions with ICON-ESM, Climate Dynamics, 61, 357–373, https://doi.org/10.1007/s00382-022-06558-w, 2023.

Raanes, P. N., Chen, Y., and Grudzien, C.: DAPPER: Data Assimilation with Python: a Package for Experimental Research, Journal of Open Source Software, 9, 5150, https://doi.org/10.21105/joss.05150, 2024.

Sakov, P. and Oke, P. R.: A deterministic formulation of the ensemble Kalman filter: an alternative to ensemble square root filters, Tellus A, 60, 361–371, https://doi.org/10.1111/j.1600-0870.2007.00299.x, 2008.

Sakov, P., Counillon, F., Bertino, L., Lisæter, K. A., Oke, P. R., and Korablev, A.: TOPAZ4: an ocean-sea ice data assimilation system for the North Atlantic and Arctic, Ocean Science, 8, 633–656, https://doi.org/10.5194/os-8-633-2012, 2012.

SALOME The Open Source Integration Platform for Numerical Simulation: http://www.salome-platform.org/, last access: 2024-08-29.

Shao, C. and Nerger, L.: WRF-PDAF v1.0: implementation and application of an online localized ensemble data assimilation framework, Geoscientific Model Development, 17, 4433–4445, https://doi.org/10.5194/gmd-17-4433-2024, 2024.

Simon, E. and Bertino, L.: Gaussian anamorphosis extension of the DEnKF for combined state parameter estimation: Application to a 1D ocean ecosystem model, Journal of Marine Systems, 89, 1–18, https://doi.org/https://doi.org/10.1016/j.jmarsys.2011.07.007, 2012.

Sluka, T. C., Penny, S. G., Kalnay, E., and Miyoshi, T.: Assimilating atmospheric observations into the ocean using strongly coupled ensemble data assimilation, Geophysical Research Letters, 43, 752–759, https://doi.org/https://doi.org/10.1002/2015GL067238, 2016.

Smith, P. J., Fowler, A. M., and Lawless, A. S.: Exploring strategies for coupled 4D-Var data assimilation using an idealised atmosphere–ocean model, Tellus A: Dynamic Meteorology and Oceanography, https://doi.org/10.3402/tellusa.v67.27025, 2015.

Strebel, L., Bogena, H. R., Vereecken, H., and Hendricks Franssen, H.-J.: Coupling the Community Land Model version 5.0 to the parallel data assimilation framework PDAF: description and applications, Geoscientific Model Development, 15, 395–411, https://doi.org/10.5194/gmd-15-395-2022, 2022.

Tang, Q., Mu, L., Goessling, H. F., Semmler, T., and Nerger, L.: Strongly coupled data assimilation of ocean observations into an ocean-atmosphere model, Geophys. Res. Lett., 48, e2021GL094 941, 2021.

Tang, Q., Delottier, H., Kurtz, W., Nerger, L., Schilling, O. S., and Brunner, P.: HGS-PDAF (version 1.0): a modular data assimilation framework for an integrated surface and subsurface hydrological model, Geoscientific Model Development, 17, 3559–3578, https://doi.org/10.5194/gmd-17-3559-2024, 2024.

The Python Language Reference: https://docs.python.org/3/reference/introduction.html#alternate-implementations, last access: 2024-02-13.

Tondeur, M., Carrassi, A., Vannitsem, S., and Bocquet, M.: On temporal scale separation in coupled data assimilation with the ensemble kalman filter, Journal of Statistical Physics, 179, 1161–1185, https://doi.org/10.1007/s10955-020-02525-z, 2020.

Trémolet, Y. and Auligne, T.: The Joint Effort for Data Assimilation Integration (JEDI), JCSDA Q, 66, 1–5, 2020.

Tödter, J. and Ahrens, B.: A Second-Order Exact Ensemble Square Root Filter for Nonlinear Data Assimilation, Monthly Weather Review, 143, 1347 – 1367, https://doi.org/10.1175/MWR-D-14-00108.1, 2015.

van Leeuwen, P. J., Künsch, H. R., Nerger, L., Potthast, R., and Reich, S.: Particle filters for high-dimensional geoscience applications: A review, Quarterly Journal of the Royal Meteorological Society, 145, 2335–2365, https://doi.org/https://doi.org/10.1002/qj.3551, 2019.

Vetra-Carvalho, S., van Leeuwen, P. J., Nerger, L., Barth, A., Altaf, M. U., Brasseur, P., Kirchgessner, P., and Beckers, J.-M.: State-of-the-art stochastic data assimilation methods for high-dimensional non-Gaussian problems, Tellus A, 70, 1445 364, 2018.

Villa, U., Petra, N., and Ghattas, O.: HIPPYlib: An Extensible Software Framework for Large-Scale Inverse Problems Governed by PDEs: Part I: Deterministic Inversion and Linearized Bayesian Inference, ACM Trans. Math. Softw., 47, https://doi.org/10.1145/3428447, 2021.

Walters, D., Baran, A. J., Boutle, I., Brooks, M., Earnshaw, P., Edwards, J., Furtado, K., Hill, P., Lock, A., Manners, J., Morcrette, C., Mulcahy, J., Sanchez, C., Smith, C., Stratton, R., Tennant, W., Tomassini, L., Van Weverberg, K., Vosper, S., Willett, M., Browse, J., Bushell, A., Carslaw, K., Dalvi, M., Essery, R., Gedney, N., Hardiman, S., Johnson, B., Johnson, C., Jones, A., Jones, C., Mann, G., Milton, S., Rumbold, H., Sellar, A., Ujiie, M., Whitall, M., Williams, K., and Zerroukat, M.: The Met Office Unified Model Global Atmosphere 7.0/7.1 and JULES Global Land 7.0 configurations, Geoscientific Model Development, 12, 1909–1963, https://doi.org/10.5194/gmd-12-1909-2019, 2019.

Whitaker, J. S. and Hamill, T. M.: Ensemble Data Assimilation without Perturbed Observations, Mon. Wea. Rev., 130, 1913–1927, 2002.

Williams, N., Byrne, N., Feltham, D., Van Leeuwen, P. J., Bannister, R., Schroeder, D., Ridout, A., and Nerger, L.: The effects of assimilating a sub-grid-scale sea ice thickness distribution in a new Arctic sea ice data assimilation system, The Cryosphere, 17, 2509–2532, https://doi.org/10.5194/tc-17-2509-2023, 2023.

Ying, Y. M.: nansencenter/NEDAS: v1.0-beta, Zenodo [code], https://doi.org/10.5281/zenodo.10525331, 2024.

Zhang, S., Liu, Z., Zhang, X., Wu, X., Han, G., Zhao, Y., Yu, X., Liu, C., Liu, Y., Wu, S., et al.: Coupled data assimilation and parameter estimation in coupled ocean–atmosphere models: a review, Climate Dynamics, 54, 5127–5144, https://doi.org/https://doi.org/10.1007/s00382-020-05275-6, 2020.

Zhao, F., Liang, X., Tian, Z., Li, M., Liu, N., and Liu, C.: Southern Ocean Ice Prediction System version 1.0 (SOIPS v1.0): description of the system and evaluation of synoptic-scale sea ice forecasts, Geoscientific Model Development, 17, 6867–6886, https://doi.org/10.5194/gmd-17-6867-2024, 2024.

Zhu, M., van Leeuwen, P. J., and Amezcua, J.: Implicit equal-weights particle filter, Quarterly Journal of the Royal Meteorological Society, 142, 1904–1919, https://doi.org/https://doi.org/10.1002/qj.2784, 2016.