JOINT CENTER FOR SATELLITE DATA ASSIMILATION

NO. 66, Winter 2020

JCSDA Quarterly

NOAA | NASA | US NAVY | US AIR FORCE

https://doi.org/10.25923/rb19-0q26

IN THIS ISSUE

NEWS IN THIS QUARTER

The Joint Effort for Data Assimilation Integration (JEDI)

Data Assimilation Challenges

All partners of the Joint Center for Satellite Data Assimilation (JCSDA) run data assimilation algorithms applied to their own models and applications. In 2001, the JCSDA was created to accelerate and improve the use of new satellite observing systems into each member's data assimilation system. As Earth-observing systems constantly evolve and new systems are launched, continuous scientific developments for exploiting the full potential of the data are necessary. Given the cost and limited lifetime of new observing systems, it is important that this process happens quickly. This effort has been successful and continues to be; but, as the context evolves, new challenges emerge.

Data assimilation algorithms are evolving and progressing to better exploit all information available. It can be argued that improvements in data assimilation methodology have contributed more to the improvements of forecast skill than the improvements in the quantity or quality of observations (Dee et al., 2014). Thus, better use of new observing systems does require access to worldwide state-of-the-art data assimilation algorithms.

Furthermore, as weather and environmental forecasting progresses, more subtle processes are taken into account. In this context, forecast models are evolving towards a more comprehensive representation of the Earth system and coupling between its components. Coupled data assimilation is desirable to better initialize coupled models, but better use of observations can also benefit from a coupled data assimilation system. The primary use case of relevance for JCSDA are the many satellite observations that are sensitive to the meteorological state of the atmosphere but also to the underlying surfaces, whether land or sea, and to the aerosols and chemical species present in the atmosphere.

Finally, models' resolutions and observation data volumes keep increasing with time, requiring more and more computationally efficient data assimilation codes. At the same time, the supercomputers where data assimilation systems are run are becoming more complex, with more and possibly heterogeneous processing elements. Using them efficiently

JOINT CENTER FOR SATELLITE DATA ASSIMILATION

5830 University Research Court
College Park, Maryland 20740

3300 Mitchell Lane
Boulder, Colorado 80301

Website: www.jcsda.org

EDITORIAL BOARD

Editor:
James G. Yoe

Assistant Editor:
Biljana Orescanin

Director:
Thomas Auligné

Chief Administrative Officer:
James G. Yoe

Support Staff:
Sandra L. Claar

is a growing concern in the community, with leading centers exploring ways to improve scalability of their forecasting models and data assimilation systems (Bauer et al., 2020). The JEDI structure evolved from the Object Oriented Prediction System (OOPS) started at The European Centre for Medium-Range Weather Forecasts (ECMWF) in the context of the scalability programme.

Data Assimilation Integration

Addressing all the challenges above, in a manner that is generic and usable by all JCSDA partners, requires a new development approach. Current operational data assimilation systems were developed 20 or 30 years ago (Kleist et al., 2009, Daley et al., 2001, Lorenc et al., 2000, Rabier et al., 2000). During that time, the fields of computing and software development have evolved tremendously, from pioneers working in laboratories or their garage to a mature industry dominating the economy. The JEDI project leverages the tools and methodologies commonly used in the software industry to address the next generation data assimilation challenges.

The first technologies leveraged by OOPS and JEDI relate to code design. Object oriented and generic programming are used extensively to separate the many aspects that constitute a modern data assimilation system. The key concept in modern software development for complex systems is separation of concerns. In a well-designed architecture, teams can develop different aspects in parallel without interfering with other work and without breaking the components they are not working on. Scientists can be more efficient focusing on their area of expertise without having to understand all aspects of the system. This is similar to the concept of modularity. However, modern techniques (such as Object Oriented programming) extend this concept and, just as importantly, enforce it throughout the code.

Data assimilation scientific papers describe algorithms with high-level notations that represent a forecast model, observation operators, or covariances matrices without relying on the implementation details of those operators. The OOPS code takes the same approach, defines abstract interfaces for the operators mentioned above, and implements the data assimilation algorithms using those interfaces. This provides separation of concerns where data assimilation specialists can focus on data assimilation algorithms without requiring a full knowledge of all underlying codes.

Separating concerns makes the data assimilation code independent from the model; it also means that it becomes generic and can be used with different models. This is the base for sharing data assimilation algorithms between JCSDA partners, significantly reducing duplication of effort. Furthermore, nothing in the OOPS code limits its application to atmospheric applications. Applications for ocean, land surface, or atmospheric chemistry can also be interfaced into OOPS. The framework caters to all components of the Earth system, enabling the evolution towards fully-coupled Earth system data assimilation.

In addition to the data assimilation algorithms, the project also makes the observation operators more generic so they can also be shared between different models through its Unified Forward Operator (UFO) and Interface for Observation Data

Access (IODA) components. As a result, any improvement in observation operators by JCSDA or any partner can immediately be shared with the other partners, without any porting or recoding. UFO and IODA make it easier and more valuable to invest into operators that will be used widely.

Joint Effort

Developing a data assimilation system for all the partners of a Joint Center, like JCSDA is necessarily a Joint Effort involving JCSDA core staff and all the partners. For collaborative work, such as code design, the software industry has matured tremendously. In this domain as well, the JEDI project makes use of modern tools and methodologies.

JEDI software development is based on an Agile approach. The principle is that development happens in small increments that are continuously integrated in the code. This is not common in numerical weather prediction or related applications but is the most common practice in the software industry as can be experienced every day with frequent updates of apps on smartphones or personal computers. The Agile approach has a number of advantages over the traditional, and dreaded, once-a-year merging of all contributions.

The most obvious advantage of working in small code changes that are merged quickly is that all developers can see and review the code as it is being developed. As a result, errors can be identified and corrected immediately and easily (it is always more difficult, even for the most experienced developers, to fix code written up to two years before than code written just a few days before). Potential conflicts with other developments can also be identified, discussed, and addressed early in the development phase rather than after the fact.

A very visible difference between JEDI and most projects in the field of weather forecasting is the level of interaction between all parties involved. The project is centered around a core team at JCSDA, with additional active in-kind contribution from partner agencies. As a result, the project team is geographically distributed. Communication technologies have progressed to a level where working on the same project across the country has become common practice in the software industry. Video-conferences involving all developers take place regularly, focusing on concrete development questions and issues. Several other working practices and cloud-based tools are used to facilitate collaborative work, including source code version control, issue tracking, continuous integration (automated testing), code reviews, and utilities for exchanging information and discussion. This is essential for working across agencies, possibly in different parts of the country, and will be used both for initial development and long-term evolution and maintenance.

Because of the distributed nature of the project, and of the future use by all partners, portability is a constant concern at every step. All JEDI code is tested automatically and with several compilers for every pull request and regularly for more extensive and expensive tests. Container technology (e.g., Docker, Singularity, Charliecloud) is used for testing and development work. Performance evaluation of code running in containers is on-going for possible use in production runs in the future, which would ease maintenance across organizations. This

effort towards portability has already been beneficial for the project: JEDI software was "ported" to cloud computing in about 20 minutes, to be contrasted with external libraries or existing models that might require months of work.

Managing a large system like JEDI with all the models from the partners, all the observation operators for all Earth system components, and all data assimilation algorithms would be impossible in a traditional monolithic code. The JEDI code is divided into components, each managed in its own repository. A flexible build system gathers the code from multiple repositories as needed by applications and builds it.

The goal of the JEDI project is to develop a worldwide state-of-the-art data assimilation system that meets operational requirements from the JCSDA partners and to make it available for research. For that purpose, the code will be released under an open source license (Apache-2). To support transition to operations and to the research community, training sessions ("JEDI Academies") are regularly organized, initially focusing on project developers and JCSDA partner agencies, progressively extending to the wider data assimilation community. Documentation is being written as the project progresses, and tutorials will be provided in the future.

The project includes the development of all the generic data assimilation components and in collaboration with the partner agencies to interface their respective models to the system. It also includes the development and maintenance of the tools necessary to develop, test, and validate the system in a collaborative environment. This will be achieved through specific tools and, where possible, the use of open source software compatible with the license of the JEDI software.

JEDI Evolution

The system includes existing leading operational data assimilation algorithms and facilitates exploration of new data assimilation science across domains and applications. Contrary to data assimilation projects in the past, such as GSI, DART, NAVDAS, or the IFS, JEDI is not centered around one single data assimilation method that would be imposed on all partners. A unified system does not mean a single configuration, and each agency will be able to use different applications and different data assimilation algorithms. It also means that JEDI will be a base for developing new data assimilation algorithms in the future and address new challenges as they arise.

Areas where all partners will benefit from JEDI improvements include, for example, efficiency and scalability on new computer architectures. Improvements in data assimilation methodology will improve the use of observations. Two foreseeable directions for research in that area are the evolution towards a fully-coupled Earth system, and the methods that can always make use of all the most recent observations available for any given forecast, regardless of their order of arrival through a more continuous data assimilation process. Finally, the JEDI UFO will encourage the common development and sharing of observation operators between the partners, collectively paving the way for the use of more observation types than any center on its own could achieve. Development of new data assimilation algorithms and improvement

of existing algorithms aiming at improving forecast skills will benefit JCSDA partners while also avoiding duplication of effort.

Final Comments

The series of articles about JEDI in this newsletter covers in more detail the main aspects of the project. The next articles explains the structure of the OOPS-JEDI code (Trémolet, 2020, this issue), followed by an article describing the interfacing with the models (Holdaway et al., 2020, this issue) and another one describing the interfacing with observations through UFO and IODA (Honeyager et al., 2020, this issue). The last article in the series describes the methodology and tools used in JEDI (Abdi-Oskouei et al., 2020, this issue). The goals of the JEDI project are very ambitious. Together, the modern practices and technologies described in these articles make the goals set forward feasible.

Authors

Yannick Trémolet (JCSDA)
Thomas Auligné (JCSDA)

References

Bauer, P., et al., 2020: ECMWF Scalability Programme - Progress and Plans, *ECMWF Technical Memorandum*, in preparation.

Daley, R. and E. Barker, 2001: NAVDAS: Formulation and Diagnostics. Mon. Wea. Rev., 129, 869–883

Dee, D.P., M. Balmaseda, G. Balsamo, R. Engelen, A.J. Simmons, and J. Thépaut, 2014: Toward a Consistent Reanalysis of the Climate System. Bull. Amer. Meteor. Soc., 95, 1235–1248, https://doi.org/10.1175/BAMS-D-13-00043.1

Kleist, D. T., D. F. Parrish, J. C. Derber, R. Treadon, W.-S. Wu, and S. Lord, 2009: Introduction of the GSI into the NCEPs Global Data Assimilation System. Wea. Forecasting, 24, 1691–1705, https://doi.org/10.1175/2009WAF2222201.1

Lorenc, A. C., and Coauthors, 2000: The Met. Office global 3-dimensional variational data assimilation scheme. Quart. J. Roy. Meteor. Soc., 126, 2991–3012.

Rabier, F., H. Jarvinen, E. Klinker, J.-F. Mahfouf, and A. Simmons, 2000: The ECMWF operational implementation of four-dimensional variational assimilation. I: Experimental results with simplified physics. Quart. J. Roy. Meteor. Soc., 126, 1143–1170.

Trémolet, Y., 2020: Joint Effort for Data assimilation Integration (JEDI) Design and Structure. *JCSDA Quarterly, 66*, Winter 2020 (this issue).

Holdaway, D., Vernières G., Wlasak M., and King S., 2020: Status of Model Interfacing in the Joint Effort for Data assimilation Integration (JEDI). *JCSDA Quarterly, 66*, Winter 2020 (this issue).

Honeyager, R., Herbener, S., Zhang, X., Shlyaeva, A., and Trémolet, Y., 2020: Observations in the Joint Effort for Data assimilation Integration (JEDI) - UFO and IODA. *JCSDA Quarterly, 66*, Winter 2020 (this issue).

Abdi-Oskouei, M., Miesch, M., and Olah, M. 2020: The Joint effort for Data assimilation Integration (JEDI) Infrastructure. *JCSDA Quarterly, 66*, Winter 2020 (this issue).

Joint Effort for Data Assimilation Integration (JEDI) Design and Structure

Introduction

Data assimilation is an essential component of any forecasting system. It aims at determining the best estimate of the state of a system given observations of the system and a previous estimate of the state of the system for use as initial condition for an ensuing forecast. It also requires error estimates of the input quantities, usually covariances matrices and bias estimates, and uses operators, such as the forecast model and observation operators.

The JEDI data assimilation system is centered around the Object Oriented Prediction System (OOPS) that defines abstract representations of the quantities and operators used in data assimilation, with the various operations that can be performed with them. This abstract interface layer is implemented in C++ using templates. Different implementations are chosen at compile time during template instantiation and provide genericity across models. Adding a new model or system to JEDI means implementing the classes defined in the abstract layer for that model or system. On the other side, adding or modifying a data assimilation algorithm is done through the interface layer so that all code at high level is model independent (***Figure 1***).

The main components of the JEDI system are described below. The description is not exhaustive and is not intended as a full technical documentation. It describes the key components and general principles behind the design. It should serve as an introduction before going deeper into the system and code.

The next section describes the abstract interfaces OOPS expects for a given model or system for which data assimilation is to be implemented. The following section shows where

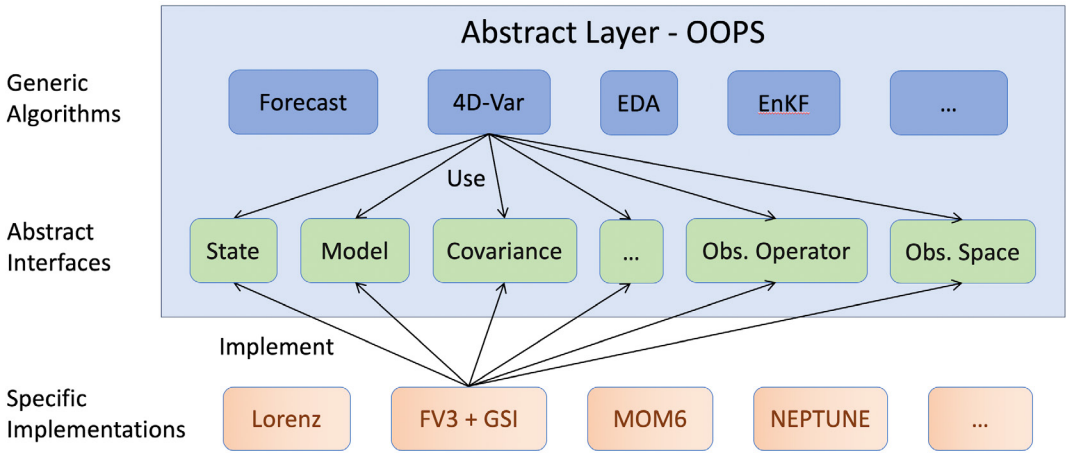

Figure 1. *OOPS generic design and separation of concerns. Applications are written in a generic layer (blue) using abstract building blocks (green), which correspond to scientific elements. Each model (red) implements the abstract elements.*

generic components can be used, and the last section gives some information about data assimilation algorithms and high-level applications.

OOPS Abstract Interfaces

Model Space Interfaces

In the scientific literature, high-level data assimilation algorithms are described in terms of a state variable x, without any reference to the specific nature of the fields in x or the geometry of their discrete representation. All such details should be encapsulated inside a state class and not be visible from high-level algorithms. In a similar way, forecast models, often denoted M(x), are used to evolve the state of the system of interest forward in time, but the details of how this is done are not required in order to define a data assimilation or other high-level algorithms, thus such details should not be apparent in high level parts of the code.

The following classes provide high-level access to model states and state related actions required by data assimilation algorithms. All assimilation algorithms refer to model space entities only through these high-level classes. The actual implementation of the classes in model space are of course model specific.

Geometry

The state of the system is obviously an important piece of data in any data assimilation and forecasting system. In meteorology and oceanography, it is represented by a collection of fields representing the values of the variables of the problem. For computational purposes, fields are discretized and a finite set of values are stored and manipulated. This set of values is distributed on a model-dependent geometry that can be relatively simple, such as regular grids, more complex, such as cubed-sphere or reduced Gaussian grids, or more abstract, such as spectral representations. The Geometry class is dedicated to holding this information in OOPS. It will typically contain the definition of the model grid, resolution, and distribution across processors. OOPS creates Geometry objects and passes them to lower level code where necessary, typically to constructors of other model space objects. Passing the same Geometry to the constructors ensures consistency in resolution and distribution across processors between the objects involved.

State and Increment

The State class is the fundamental class giving access to operations on model states in OOPS. It holds and encapsulates data that define the state of the system and the date and time for which it is valid. In addition to encapsulating data, the State class provides operations associated with that state, such as basic utility functions (read, write, diagnostic prints), and a method (getValues) to provide access to state values where necessary.

The Increment class is very similar to the State class except for the fact that it handles perturbations to the state. It provides a method to compute an increment as the difference between two states, to add an Increment to a State, and a set of basic linear algebra operators, which are legitimate operations for increments but not for states.

Model

The main responsibility of the Model class is to hold the forecast model configuration data and to provide the ability to evolve a State in time. Its main method is the forecast method, which is coded in the OOPS layer and

relies on specific model implementations to provide three lower level methods: initialize, step, and finalize. For a given model, these three methods contain respectively everything that happens before the loop over time steps, inside the loop, and after the loop. Although these do not necessarily exist as such in every model, they are fairly easy to create by wrapping existing code. For improved efficiency in the case several calls to the forecast are made in the same executable, code that should be executed only once should be in the constructor of the Model, while code that needs executing at the beginning of each integration should be in the initialize method.

The reason for this design of the abstract interface for the model is that 4D assimilation methods require access to the model state throughout the assimilation window. The volume of data that would represent far exceeds the amount of memory available on supercomputers, so it is not possible to store it in memory. For the same reason, I/O would be prohibitively expensive. These four-dimensional computations have to happen while the model is running. Some algorithms also require running the forecast and computations needed for data assimilation repeatedly in an iterative process. The level at which the interface between the model and the data assimilation is written in OOPS is the highest level possible that meets those requirements while remaining generic. Applying the same object-oriented approach deeper into the models would be possible, and possibly desirable in terms of code design for the future, but it is not needed for data assimilation and is not in the scope of OOPS or JEDI.

PostProcessors

In addition to the initial condition and length of the required forecast, a PostProcessor object is passed to the Model's forecast method. The PostProcessor class is a very generalized form of post-processing used to handle all output a forecast should produce while running. The goal of that class is to isolate all the auxiliary code that should be called during the integration of the forecast model but is not strictly part of the model (enforcing separation of concerns).

PostProcessors derive from a base class that controls when each processor will be called during the forecast execution based on the configuration passed as an argument to its constructor. This relieves both the forecast model and the actual processor from that responsibility, allowing them (and more importantly, developers) to focus on their responsibilities, which are to produce respectively a forecast and some output given the model state generated by the forecast.

Concrete post processors define a process method taking a State or Increment as unique argument that cannot be modified. In addition to this, each processor class can implement complex constructors and/or destructors or other methods to be called before and/or after the forecast run as necessary. For simple cases, this will not be useful (for example, to print norms to a log file or to save the forecast to files). In other cases, complex operations can be implemented, for example setting-up observations operators before the forecast starts in the processor's constructor, calling them during the forecast integration from the mandatory process method and

collecting observation equivalents after the forecast has completed in a specific method. Storing the "trajectory" for tangent linear and adjoint models is also the responsibility of a dedicated post-processor.

In variational data assimilation, this approach is important as all the contributions to various terms of the cost function need to be collected in the same model run while keeping a lot of flexibility in implementing new terms, and keeping the data assimilation code isolated from the model code.

Other Classes in Model Space

A few other interface classes exist in model space; for example, auxiliary classes for state augmentation for applications such as parameter or model bias estimation. A LinearModel class also exists to hold tangent linear and adjoint models.

Observations Space Interfaces

In the scientific literature, high-level data assimilation algorithms are described in terms of a vector of observations, usually denoted y, without any reference to the specific nature of the observations in y or their distribution in space and time. All such details should thus be encapsulated inside an Observations class and not be visible from high-level algorithms. In a similar way, observation operators, often denoted H(x), are used to simulate the observations given the state of the system x, but the details of how this is done are not required in order to define a data assimilation or other high-level algorithms; thus, such details should not be apparent in high-level parts of the code.

The following classes provide high-level access to observations and observation-related actions required by data assimilation algorithms. All assimilation algorithms refer to observation space entities only through these high-level classes. The actual implementation of the classes in observation space are specific to each observation type.

Observations and Departures

The Observations class holds and encapsulates observations (often denoted y in the scientific literature) and associated operations. The Departures class is the mirror in observation space of the Increment class. It represents differences between, or perturbations to, observations. It is very similar to the Observations class except for the fact that it provides an additional set of linear algebra operators.

For data assimilation applications, there is one object of class Observations per observation type. There is no strong constraint in the system about what constitutes an observation type other than the fact that the same observation operator, quality control procedures, and bias correction method will be applied to all observations in a given type.

The Observations and Departures classes are implemented in OOPS and are not part of the interface to the actual model implementation. Internally, Observations and Departures objects each contain a vector of values (ObsVector) to which methods are delegated.

ObsSpace and ObsVector

The concept of geometry from the model space is transposed to the observation space with the ObsSpace class. This class defines the distribution of observations in space, as the Geometry would for model space entities. However, the ObsSpace class also gives access to all metadata associated with

the observations, including time, some instrument dependent metadata, quality control information, and to the actual observation values themselves.

The ObsVector class is used to hold values in observation space. Like State and Increment are always defined with a Geometry, ObsVectors always refer to an ObsSpace. The values in an ObsVector can be read from the ObsSpace, for example, the observations values, or can be local variables computed within the data assimilation process, for example H(x). In that case, the values in the ObsVector can be saved in the ObsSpace for later diagnostics or can be deallocated like any other local variable.

Optimisation of data access, for example, according to frequent access patterns or according to specific observation distributions across processors, are left to the lower-level implementation of each model, as it is impossible to define optimisations that would suit all possible applications.

ObsOperator

The computation of a simulated observation given a model state comprises two steps: the interpolation of the model variables to the observation locations, and the computation of the observation equivalent from these interpolated model variables. It is the responsibility of the state to provide the values of its variables (or fields) at the requested locations through the getValues method, which isolates the observation part of the code from the internal geometry of the model being used.

The ObsOperator class define the actual computation of observation equivalents given model state values at the observations' locations for a given observation type (method simulateObs). This step can be extremely simple if the quantity being measured is a variable of the model (a temperature or wind measure for example), or very complex in the case of radiance observations from satellites involving a radiative transfer model. In any case, this class encapsulates the science related to the observation type and isolates it from the technical details related to the forecast model.

The LinearObsOperator class also exists and holds the tangent linear and adjoint of the observation operator.

Other Classes in Observation Space

An auxiliary observation control variable and the corresponding increment are defined in the interface directory. This is intended for applications that require control of parameters in the observation operators or for estimation of observation bias. A covariance matrix for these parameters is required.

Error Covariances

Background Errors

Modelling background error statistics is an essential part of a data assimilation system and a key element to the quality of the analysis and ensuing forecast. The design of the interfaces for background error covariance matrices is, however, very simple. In addition to a constructor, the ErrorCovariance class requires a method to multiply an increment by the covariance matrix and another one to multiply an increment by the inverse of the covariance matrix. In real cases, this inverse is often very ill conditioned, the variational algorithms in OOPS only use it for diagnostics. Some applications require a method to generate a

random increment according to the statistic distribution described by the covariance matrix, this is particularly useful for testing.

OOPS implements a factory that lets users choose a given covariance model at run time. Generic background error covariance implementations are implemented in the System Agnostic Background Error Representation (SABER) repository.

Observation Errors

Just like the background errors, the definition of observation errors is key to the quality of the analysis, but the interface to the covariance matrix is simple and mimics that of the background error covariance. The type of observation error covariance is configurable at run time, although initially only the diagonal observation error covariance is implemented. More generic covariance models will be developed. For example, generic components developed in the context of background error covariance modeling to represent correlations on unstructured grids open the possibilities for more advanced observation error correlations representations.

Other Interface Classes

Locations

The Locations class is one of the links between the observation and the model parts of the code. It is used to specify the locations where model fields values are needed by the observation operators and passed to the getValues method of the state. OOPS only passes Locations objects between methods associated with other classes and makes no assumption regarding the interface of that class. Different applications can use that flexibility to implement very different coordinate systems and specific methods.

GeoVaLs

The GeoVaLs (Geophysical Values at Locations) class is used to pass the model values interpolated to the required locations to the ObsOperator (***Figure 2***). The GeoVaLs class is the other link between the model and the observation parts of the code. The only use of this class in OOPS is to pass the model fields values at the locations of observations from the getValues method to the observation equivalent computation. As a result, there are no constraints regarding the interface of this class in a given system.

Traits

The set of classes that represents the system of interest is gathered as a list of aliases (typedef in C++) that define the concrete class that corresponds to each abstract class expected by OOPS. This list takes the form of a C++ class called a trait. The OOPS code is generic and templated on this trait. At compile time, through template instantiation, the compiler replaces the generic classes in OOPS by the specific classes defined in the trait. This

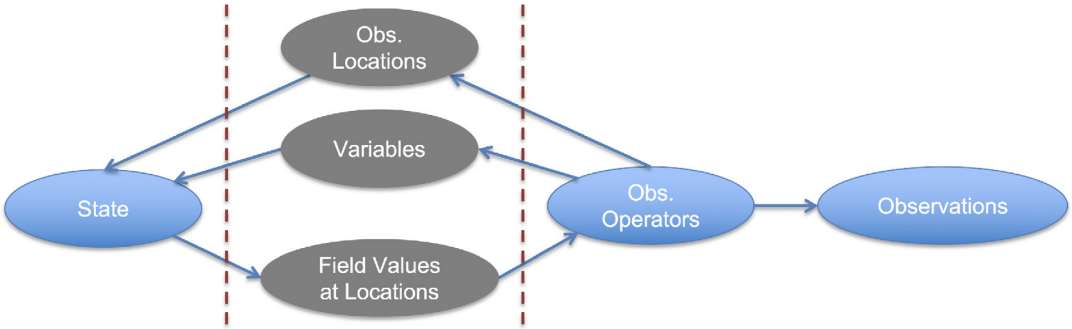

Figure 2. *Interactions between models and observations in JEDI. The GeoVaLs class is used to handle the state values evaluated at the locations of the observations.*

defines a generic data assimilation system that can be used with many models without re-writing any code.

Templates do not exist in Fortran and no equivalent functionality is expected in the future. It is this functionality, more than the object-oriented aspects, that drove the choice of C++ as a programming language for OOPS and JEDI.

Comments About Interface Classes Design

In model space, the State and Increments are the classes that make the interface between the abstract layer and the concrete implementation. In observation space, Observations and Departures are implemented in OOPS, it is the lower level ObsVector that constitute the interface. A similar approach could have been chosen in model space with a concept such as Fields for example. However, fields in model space are a representation of a continuous reality, for which the concepts of resolution and change of resolution exist. Even more than that, fields can be represented in different spaces (grid point, spectral, finite volume, spectral elements, etc.). Even in the same system, a linear model can be used jointly with a nonlinear model that does not use the same representation for fields (perturbation forecast model). In observation space, the observations are discrete and their number finite. There is no concept of change of resolution or different representation. Thus, it is reasonable to have only one data structure and only one class to interface; whereas, this would have limited possibilities in model space.

The classes that define a specific system in a given trait are wrapped in an interface layer in OOPS. This is not mandatory and was not done in earlier versions of the code. This interface layer was added to help readability. In particular, when adding a new model into the system, this layer provides a single point where all interfaces are visible. Without it, one would have to scan all the code to see where the classes in the trait are used to have a complete list of methods and their interfaces for each class. Such a list could be maintained in the documentation but would inevitably go out of date. By being part of the code, it is permanently checked by the compiler and cannot go out of date. It is a case where the code really is the documentation. The interface layer is also used to instrument the code. It contains utilities for tracing and timing the execution of the code. Such tools could be extended in the future.

Generic Components

Each model interfaced with OOPS requires its own trait with a complete set of consistent classes matching the expected abstract interfaces. However, a given concrete class can be used in more than one trait. This opens possibilities for sharing more code across different models than just the high-level data assimilation code, increasing collaboration and reducing duplication of effort even further. This is where JEDI extends OOPS with more generic components (***Figure 3***) in UFO, IODA and SABER.

The observation-related generic layer (UFO and IODA) reduces code duplication for observation handling and functionality that is common to all observation types, including quality control procedures and bias correction. While the UFO and IODA classes are specified in the OOPS trait and chosen at compile time, specific operators within UFO are chosen at run time, bringing another level of flexibility to the system.

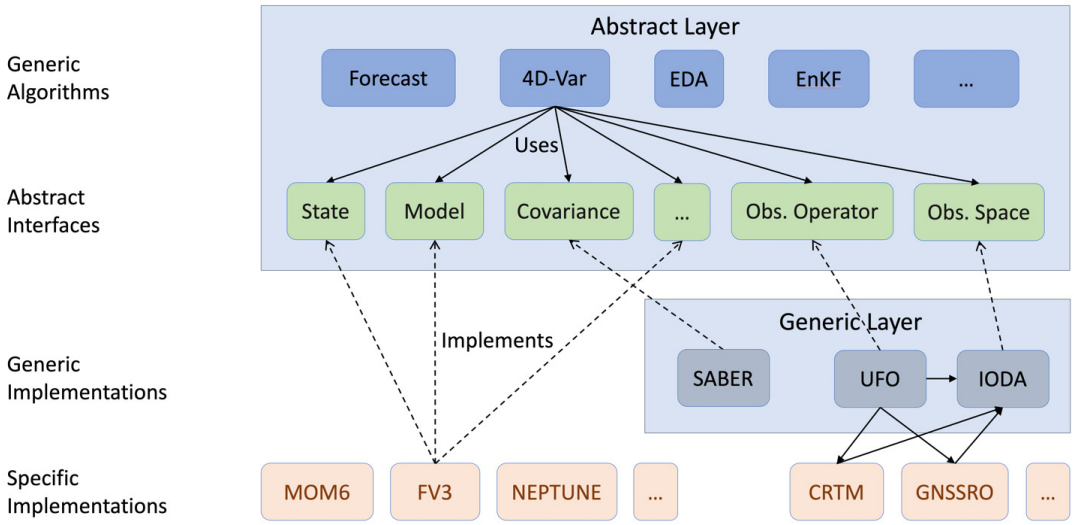

Figure 3. *JEDI Structure with a generic layer for observation and background error handling.*

UFO

The Unified Forward Operator (UFO) is at the heart of JCSDA's mission and is the other major component of JEDI after OOPS. It implements generic observation operators. The key elements that make the observation operators generic are the classes that make the connection between the model space and the observation space. As described above, these classes are Locations and GeoVaLs. Thus, UFO includes implementations of these two classes in addition to a collection of observation operators that users can leverage. The same UFO operators can be used in conjunction with several models regardless of the models' internal geometries.

In addition to the observation operators themselves, UFO also implements generic tools for quality control and variational bias correction, which are extremely important for operational centers. These aspects represent a large fraction of the work related to observation in operational centers and thus have a lot of potential for reduction of duplication of work. Once a UFO generic operator has been validated, including it in any JEDI-compliant system should be as easy as installing a new app on a smartphone from an App store. Generic operators will be provided with default settings that can be fine-tuned for each application.

IODA

The Interface for Observation Data Access (IODA) is developed with the UFO to handle observation data. It provides functionality for I/O of observation data and in memory access. In the interface layer and traits, it implements the ObsSpace and ObsVector classes.

IODA will facilitate the implementation of UFO. It will also facilitate the exchange of observations between centers for experimental studies, comparisons, or potentially for reanalyses. Data assimilation diagnostics will also be developed based on IODA.

SABER

The System Agnostic Background Error Representation (SABER) implements generic background error covariance matrices, among which is BUMP (Background error on Unstructured Mesh Package). After observation processing, the modelling of background errors is the most

time-consuming task in data assimilation. Exchange and comparisons of covariance matrices have never been possible before JEDI. This will be an important tool for scientific advances in data assimilation.

Comments

Each component of JEDI is stored in its own code repository. (***Figure 4)*** It is the build system that gathers all code for a given build. This approach enforces separation of concerns and gives additional flexibility to build applications with the components it requires. It also facilitates development with different teams managing each repository.

Data Assimilation and Other High-level Algorithms

Data assimilation algorithms in OOPS-JEDI are entirely written using the abstractions described in the previous sections. At this time, most existing variational data assimilation algorithms including 3D-Var, 3D-FGAT, 4D-Var, 4D-En-Var, weak constraint 4D-Var, and observation space algorithms are implemented in the OOPS layer. Ensemble data assimilation algorithms (EDA and EnKF-based) are being developed. Minimization algorithms for variational data assimilation are written in a very generic manner and can be reused for other purposes. For example, an approximate inverse of the background error covariance matrix can be computed using one of those minimization algorithms for cases where no specific inverse is implemented in an ErrorCovariance model.

Algorithms related to data assimilation, for example, the computation of observation impact, are also implemented in the high-level generic layer.

Although already covering most operational data assimilation algorithms, the data assimilation layer will evolve in the future. New research will take place and new algorithms will be developed that can be immediately tested with all models interfaced with JEDI, from a very simple Lorenz model to the most advanced fully-coupled Earth system models.

Authors

Yannick Trémolet (JCSDA)

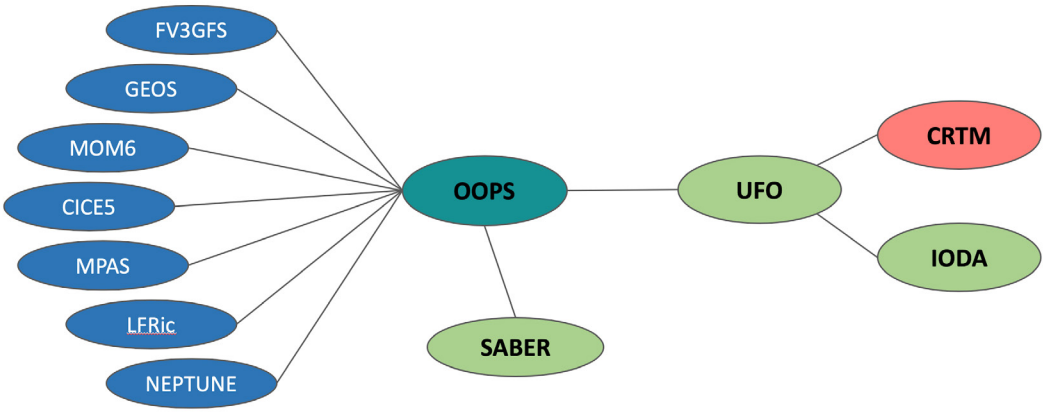

Figure 4. *Each major component of JEDI is managed in its own repository.*

Status of Model Interfacing in the Joint Effort for Data Assimilation Integration (JEDI)

Introduction

As outlined in the introductory articles, the central components of JEDI are completely generic in the sense that different data assimilation systems can make use of the same software infrastructure and components. This is achieved through so-called interface classes that define how the various components will interact with each other. However, some work is still required to build all the variables and methods of the interface classes for some specific forecast model so that the applications of JEDI can be used with that model. Examples of interface classes implemented for specific forecast models include Geometry, State, Increment, Model, LinearModel, and VariableChange. All of the operations in these interface classes depend on the grid of the underlying model or the model itself.

Generic tests in Object Oriented Prediction System (OOPS) confirm that the interface classes have been developed correctly and, once complete, applications based on the algorithms provided by OOPS can be created and used. Most of the actual models are written using Fortran, rather than C++, so the interface classes can just call into some Fortran equivalent. Several models are now interfaced to JEDI or are in the process of being interfaced. These are outlined in ***Table 1***.

Table 1. *Models for which an interface to JEDI has been or is being prepared.*

MODEL	TYPE	GRID	INTERFACE REPOSITORY	CENTER
GFS	Atmosphere	Cubed-sphere	FV3-JEDI	NOAA-EMC
GEOS	Atmosphere	Cubed-sphere	FV3-JEDI	NASA-GMAO
GFS GSDChem	Aerosol	Cubed-sphere	FV3-JEDI	NOAA-ESRL
GEOS-AERO	Aerosol	Cubed-sphere	FV3-JEDI	NASA-GMAO
LFRIc	Atmosphere	Cubed-sphere	LFRIc	MET Office (UK)
MPAS-A	Atmosphere	Voronoi meshes	MPAS-JEDI	NCAR
NEPTUNE	Atmosphere	Cubed-sphere	NEPTUNE	NRL
Quasi-geostrophic	Toy model	Lat-Lon	OOPS	ECMWF
Lorenz 95	Toy model	1D	OOPS	ECMWF
ShallowWater	Toy model	Lat-Lon	shallow-water	NOAA-ESRL
MOM6	Ocean	Tripolar	SOCA	NOAA-EMC
SIS2	Sea-ice	Tripolar	SOCA	NOAA-EMC
CICE6	Sea-ice	Tripolar	SOCA-CICE6	NOAA-EMC
WRF	Atmosphere	Lat-Lon	WRF-JEDI	NCAR

The generic nature of JEDI provides a great deal of flexibility as the complexity of the system increases. Rather than large monolithic software, there can exist smaller components that cover specific tasks or sets of tasks. Model interfaces are constructed for individual components of the Earth system (e.g., the atmosphere or ocean). There is no explicit need to combine different components into one interface, meaning they can be updated independently as models change. In addition, it does not limit scalability of development, an important consideration for a community effort of this magnitude.

OOPS provides the methods for creating forecasts from within the data assimilation algorithms and in the same executable. This is important where 4D data assimilation is used with outer loops, since it avoids reading and writing large files and cuts down the number of model initializations that are required. There is a forecast model class within OOPS for handling initialization, stepping, and finalization of model forecasts.

Status of the Model Interfacing

The sections below outline the efforts ongoing for the interfacing of JEDI to the next generation forecast models being used at various centers in the United States and internationally. The details of interfacing to the toy models is omitted since these mostly exist to provide fast regression testing and interfacing examples rather than scientific capability. Details of the MPAS model are also omitted since these were covered in a previous edition of this newsletter McMillen (2019). Most of the models being interfaced to the JEDI system are so-called next generation models. They are typically non-hydrostatic and use grid construction that avoids the converging of grid points seen in longitude-latitude grids. In general, the results below demonstrate infrastructural capability since that is the stage of development for most interfaces. Focus is currently shifting to examining the scientific validity of high-resolution results and cycled experiments.

FV3-JEDI

The finite volume cubed-sphere (FV3) dynamical core (Lin 2004), developed by NOAA's GFDL laboratory, is used in NASA's Goddard Earth Observing System (GEOS) model and now NOAA's Global Forecast System (GFS) model. It is a non-hydrostatic model that uses cubed-sphere geometry, finite volume dynamics, and a Lagrangian vertical coordinate. FV3 also supports a two-way nesting capability, which will be used for regional modeling efforts at NOAA.

The FV3-JEDI interface, named for the model component that governs the horizontal grid, is being built to provide JEDI-based data assimilation for all models, both global and regional, that use the FV3 dynamical core. By having a single interface, it will help bolster collaboration between and within centers that use FV3-based models and eliminate duplicate effort. In addition to providing the meteorological data assimilation, it will also provide the mechanism for doing other kinds of atmospheric data assimilation, such as aerosol, chemistry, and constituent.

FV3-JEDI is interfaced to four different FV3-based model drivers: MAPL, which drives the GEOS model; NEMS, which drives the GFS model; FV3-JEDI-LM, which drives a stand-alone FV3 dynamical core; and a 'pseudo' model, which can read a previously-produced forecast from either GEOS or GFS.

For the simplified FV3-JEDI-LM system, the model can be driven by FV3-JEDI, exchange states with JEDI, and be rewound for outer loops. For the MAPL and NEMS interfaces, FV3-JEDI can drive the forecast model and retrieve states as the model advances. NASA's Global Modeling and Assimilation Office (GMAO) has developed adjoint and tangent linear versions of FV3, as well as the GEOS convection, cloud, and turbulence schemes. This linearized model is available through the FV3-JEDI-LM repository. Having this linearized model on the FV3 grid enables 4DVar data assimilation with FV3-JEDI, as well as adjoint-based Forecast Sensitivity Observation Impacts (FSOI).

With FV3-JEDI, all common flavors of data assimilation have been tested: 3DVar, 3DEnVar, 4DEnVar, 3D-FGAT, and 4DVar, as well as their hybrid equivalents. The formulation of the B matrix uses BUMP, so far with prescribed length scales for correlation and localization. The static part of the B matrix uses a univariate formulation applied to stream function, velocity potential, temperature, relative humidity, and surface pressure. Work is underway to extend the static B matrix formulation to include more sophisticated balance operators and thus a multivariate formulation. FV3-JEDI has been interfaced to in-situ temperature, wind and humidity observations, radiances modeled with the Community Radiative Transfer Model (CRTM), bending angle observations using the Global Navigation Satellite System Radio Occultation (GNSS-RO), surface observations, and aerosol optical depth (AOD) modeled with the CRTM.

Figure 1 shows the analysis increment of the eastward component of wind using the 4DEnVar assimilation procedure. The increment is plotted at around 150hPa, the approximate height of the jet stream. The horizontal resolution of the grid is around 50km (C180 in FV3 terms). The background comes from the GEOS model.

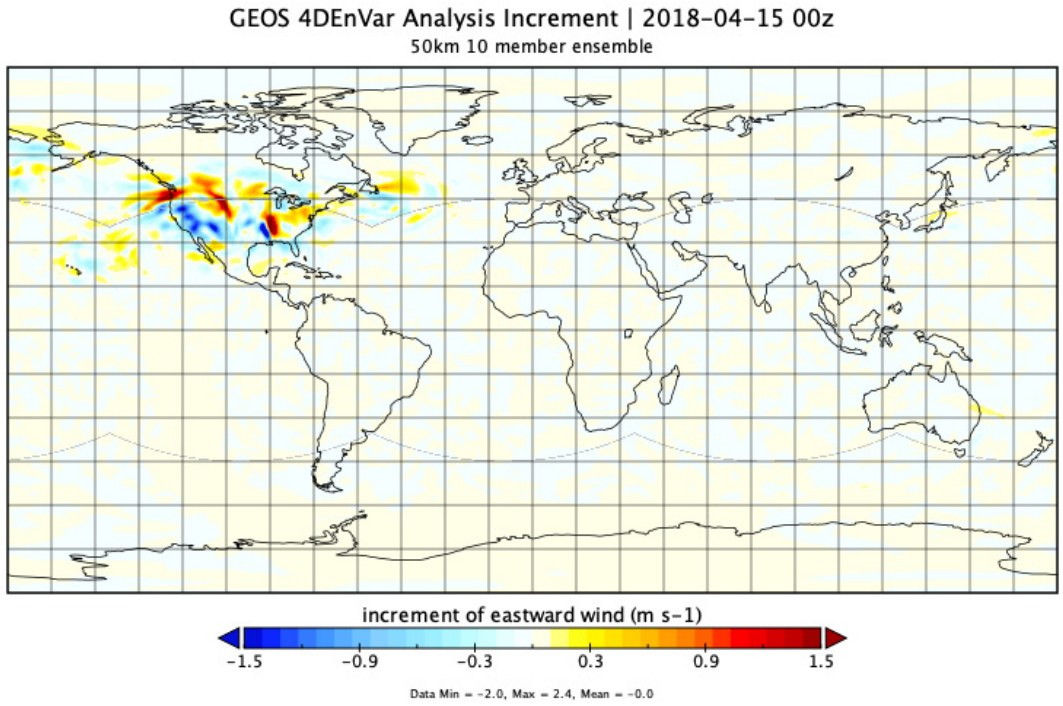

Figure 1. *Analysis increment of the eastward component of wind at ~150hPa valid at 2018-04-15 00UTC. 4DEnVar data assimilation using the FV3-JEDI interface, assimilation of in-situ radiosonde, and aircraft temperature and wind. Localization handled using BUMP and defined using prescribed length scales.*

In this experiment, a subset of the in-situ radiosonde and aircraft wind and temperature observations are assimilated. Most of these are over the United States. The localization is computed using BUMP and a prescribed length scale of around 1000km. The increment shown in the figure is plotted directly on the cube sphere grid, outlines of which are shown on the figure due to an artifact of the plotting software used. When computing the forward operator, the interpolation is directly from the native cubed-sphere grid to the observation locations. The assimilation window is 6 hours centered on 2018-04-15 00z. Despite this being a fairly simple experiment, there are clear dynamical features in the increment, and relatively smooth and realistic structures are seen.

Figure 2 shows a similar experiment to that shown in Figure 1 except using the 4DVar algorithm and a hybrid formulation of the B matrix. The static part of B is univariate and based on prescribed, rather than dynamically computed, length scales. This figure shows temperature lower down in the troposphere at around 850hPa. The tangent linear and adjoint model used in the 4DVar uses moist physics and boundary layer in addition to the dynamical core. The dynamical influence, through the tangent linear and adjoint of the forecast model, results in large scale increment that spreads away from the observations and follows the dynamical trajectory of the atmospheric flow.

The results above rely on a simple formulation of the static background error covariance so little weight is given to it. Modeling static background error covariances on the native model grids presents a number of challenges. Given the computational expense of representing multivariate covariances, the traditional approach relies on first transforming to unbalanced variables so covariance can be modeled univariately. For

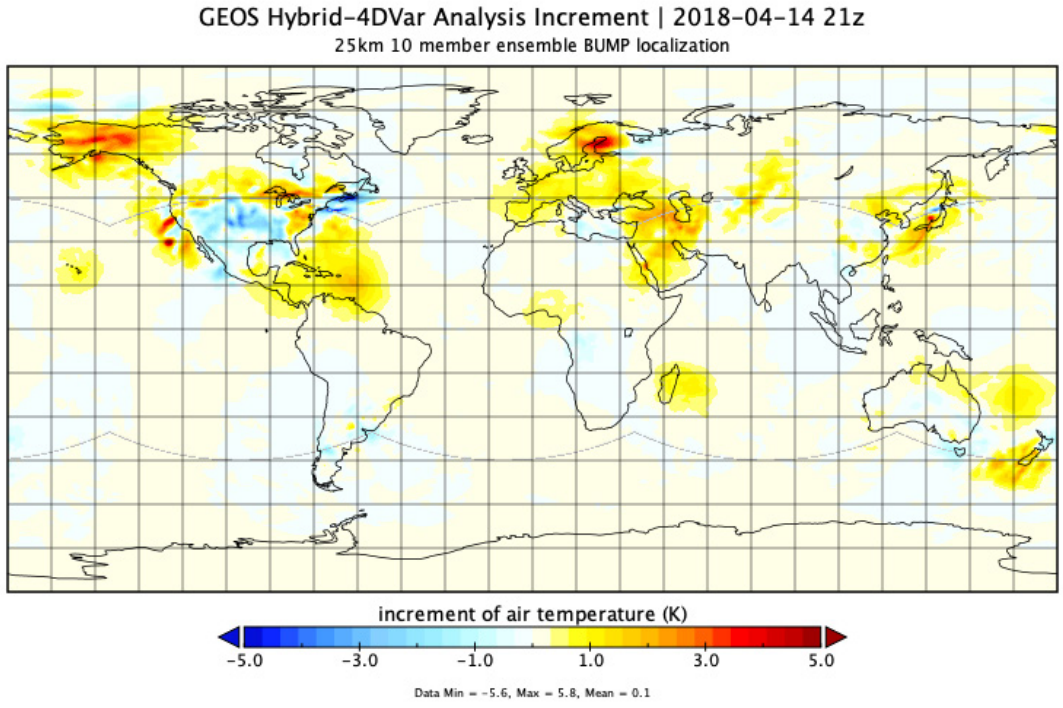

Figure 2. *Analysis increment of temperature at ~850hPa valid at 2018-04-14 21UTC. Hybrid-4DVar data assimilation using the FV3-JEDI interface, assimilation of in-situ radiosonde, and aircraft temperature and wind. Univariate correlation and localization handled using BUMP.*

meteorological applications, this involves converting to stream function and velocity potential variables and then using statistical regression to solve the linear balance equation. Converting from winds to stream function and velocity potential requires solving an inverse Laplacian Poisson problem, something that is relatively inexpensive on a latitude-longitude grid but more challenging on a cubed-sphere grid. For FV3-JEDI, a Finite Element multigrid solver has been assembled based on code provided by Cotter and Thuburn, 2014. The majority of the algorithm is generic except the prolongation and restriction operators, which are currently limited to cubed-sphere and icosahedral hexagonal grids. ***Figure 3*** shows the D-Grid tangential winds before and after application of the Poisson solver. The winds, which are on a 100km (C96) cubed-sphere grid, are converted to stream function and velocity potential and then converted back to winds using a straightforward derivative. The solver is run with a 5-grid hierarchy. The recovered winds are very close to the starting winds, showing the solver to be working. Small discrepancies come from the need to average the winds from the D-grid to the C-grid and back during the algorithm. Note the discontinuities at cube edges. This is due to D-Grid winds being grid tangential, i.e. aligned with the grid boxes, while indexing on cube faces begins at different corners to allow seamless joining of faces. This results in the u components being perpendicular across some face interfaces. Work is now underway to train a BUMP-based covariance model for univariate unbalanced variables so the above experiments can be repeated with a more sophisticated modeling of the covariances.

Figure 4 shows the dust (bin 1) increment for the NOAA Global Systems Division Chemistry (GSDChem) model. The increment is produced using a 3DEnVar algorithm (i.e., where the B matrix model is produced only from the ensemble). The increment is produced with the same FV3-JEDI interface that is used for the above meteorological experiments and using the same executable. This level of flexibility is enabled by a design that lets users make run time choices of variables. The observations are from Suomi-NPP 500nm AOD and the forward operator used is the CRTM. The experiment is performed for a relatively coarse 200km horizontal grid (C48) and with a 6-hour assimilation window. The increment appears over and downstream of the Sahara Desert where it is expected.

SOCA

The Sea-ice Ocean Coupled data Assimilation (SOCA) project is a broad effort to deliver coupled marine data assimilation for NOAA that leverages the JEDI infrastructure. The project goes beyond building a model interface to the individual marine components to also assembling generic marine observation operators and data handling. The model components that SOCA is interfacing to are the Modular Ocean Model version 6 (MOM6) ocean model (Adcroft and Hallberg 2006) and the Community Ice Code (CICE) sea-ice model (Walters, et al. 2015). Like FV3, MOM6 is a GFDL model; CICE is developed by the Department of Energy. NASA's GMAO is also planning on implementing the MOM6 ocean model into GEOS and leveraging the interfacing work of SOCA.

An extensive set of observations have been interfaced to in SOCA; these are outlined in ***Table 2***. Note that the nonlinear, tangent linear, and adjoint versions of all operators are available, making variational data

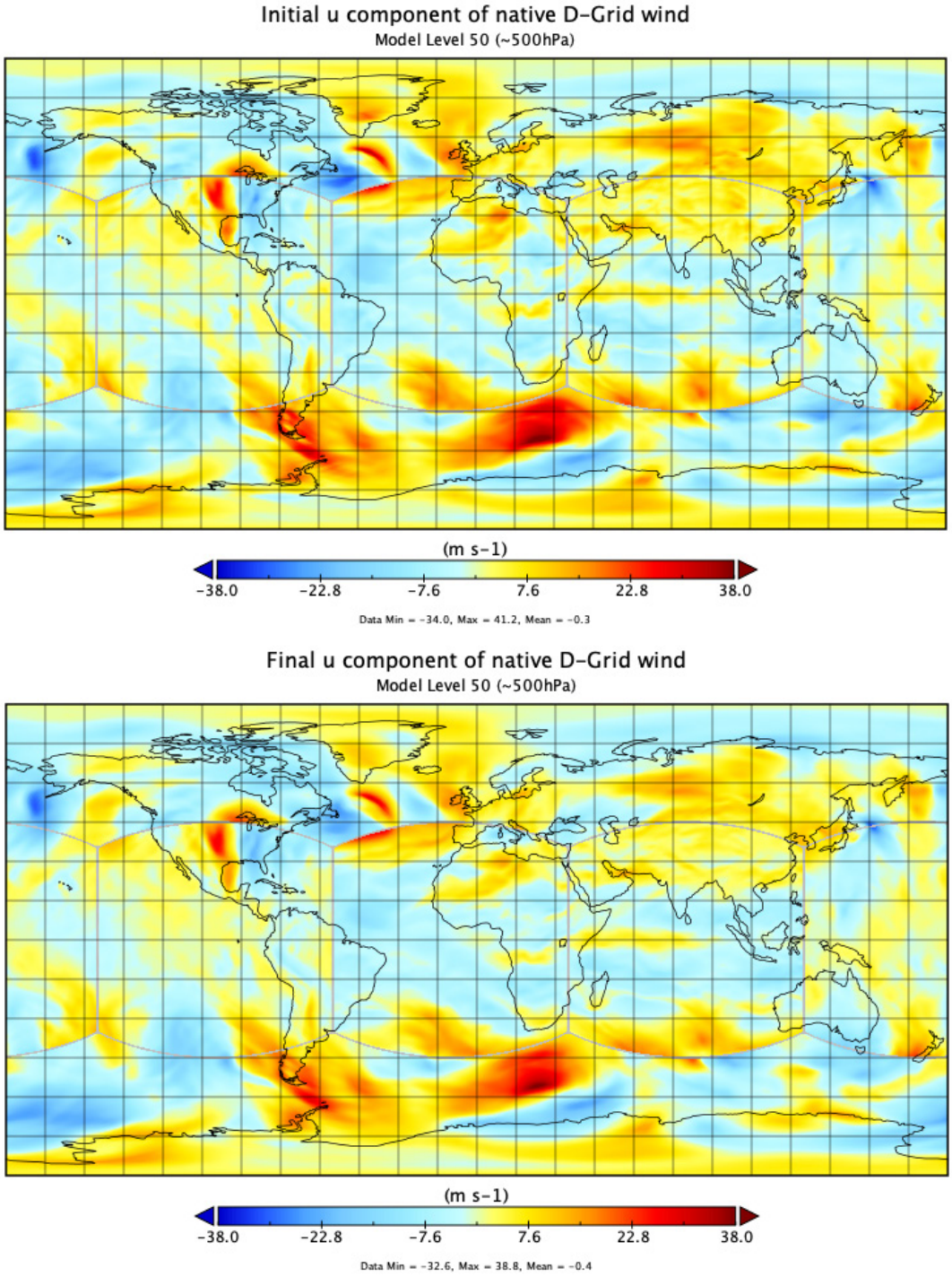

Figure 3. *U component of the D-grid tangential winds before (top) and after (bottom) the application of a Poisson solver to convert them to stream function and velocity potential and then back using a straightforward derivative.*

assimilation algorithms possible for all marine models that are or will be interfaced to JEDI.

The SOCA interfacing has been tested with 3DVar, 3DEnVar, and 3D-FGAT data assimilation and their hybrid variants for MOM6 and CICE. An interface to the stand-alone MOM6 forecast model has been implemented, making it possible to drive the model and retrieve states. This makes, for example, 3D-FGAT and cycling possible

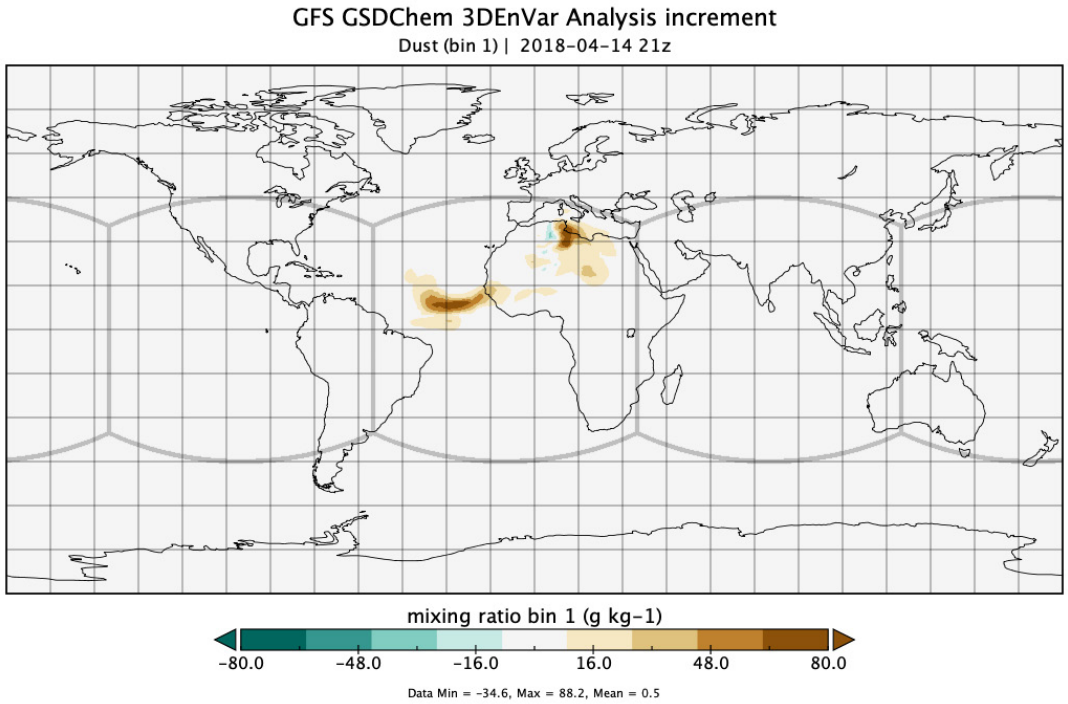

Figure 4. *Analysis increment of dust bin 1 at ~700hPa valid at 2018-04-14 12UTC. 3DEnVar data assimilation using the FV3-JEDI interface, assimilation of AOD using the CRTM. Localization handled using BUMP.*

Table 2. *Summary of the observation types implemented in UFO and used in a typical SOCA assimilation cycle.*

RETRIEVED QUANTITY	SENSOR/SATELLITE	THINNING RATE	TYPICAL COUNT ASSIMILATED
Sea Surface Temperature from Infrared	AVHRR – NOAA-19	99.5%	110,000
	AVHRR – METOP-A	99.5%	150,000
	VIIRS – NPP	99.5%	250,000
	ABI – GEOS-16	Monitoring Only	Monitoring Only
Sea Surface Temperature from Microwave	GMI – GPM	75.0%	110,000
	AMSR2 – GCOM-W1	75.0%	130,000
	WindSat	75.0%	100,000
Sea Surface Salinity	SMAP Radiometer	0.0%	450,000
Absolute Dynamic Topography	Jason-2	0.0%	240,000
	Jason-3	0.0%	240,000
	Sentinel-3a	0.0%	240,000
	Cryosat-2	0.0%	240,000
	SARAL	0.0%	240,000
Ice Fraction from Microwave	F17 & F18	95.0%	100,000
			Total: 1,640,000

all using the SOCA interface. In addition, it is possible to read model states in a pseudo model mode so SOCA can be interfaced to the full NOAA coupled model. A high-resolution quarter degree cycled experiment has been conducted with SOCA updating the ocean and sea-ice states, but with the forecast model being the fully coupled FV3-MOM6-CICE5. In the experiment, the forecast states are read in a pseudo model mode. ***Figure 5*** shows the increments of sea surface height and sea surface temperature from within the cycling. All the observations listed in Table 2 are used in the assimilation.

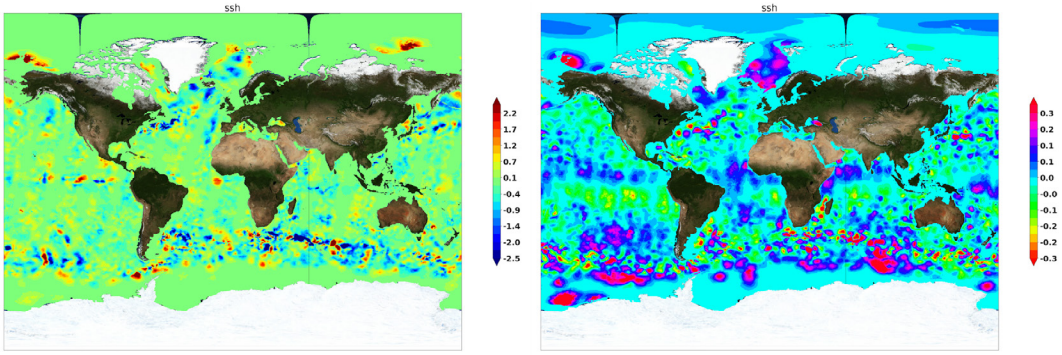

Figure 5. *Increments of sea surface height (left) and sea surface temperature, valid at 2013-10-01 00z (right).*

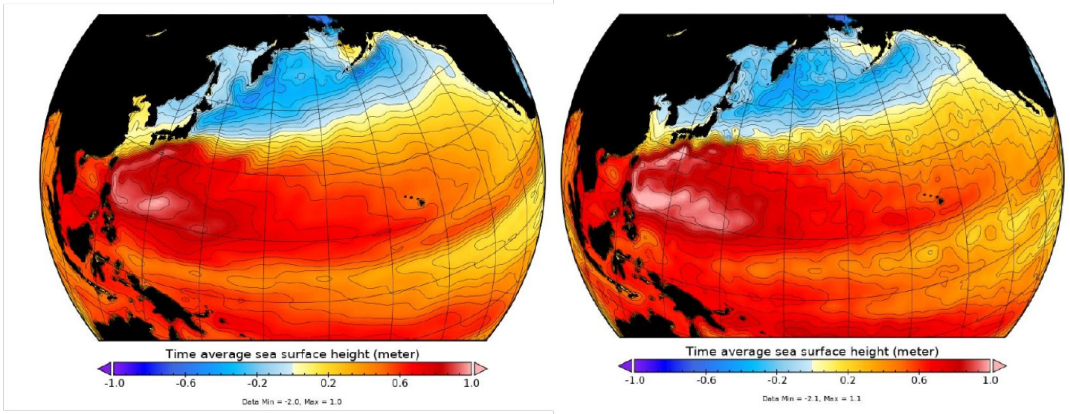

Figure 6. *SOCA cycled 3Dvar results. Comparison of sea surface height on day 30 of a 30-day free run of the MOM6 ocean model (left) and a 30-day cycled 3Dvar (right).*

A 30-day cycling for MOM6 has been conducted with the SOCA interface. This was done using 3Dvar and a 24-hour assimilation window. The horizontal grid spacing used in the experiment is one degree (approximately 100km). A total of 1.6 million observations were assimilated per cycle; observations assimilated were satellite sea surface temperatures (NESDIS/ACSPO AVHRR L2P) and altimetry (Jason-2, Jason-3 Sentinel-3a, Cryosat-2 and SARAL). ***Figure 6*** shows the sea surface heights on the final day of the cycled experiment. It is clear that more detail is present in the sea surface height field in the run that included data assimilation. In particular, the Kuroshio meander east of Japan has much more detail and realistic structure.

Neptune

The Navy Environmental Prediction System Using the NUMA Core (NEPTUNE) is the next generation model being developed by the Naval Research Lab (NRL). It uses the Nonhydrostatic Unified Model of the Atmosphere (NUMA) dynamical core (Giraldo, et al. 2013), which is a spectral element system. NEPTUNE uses a cubed-sphere grid, though NUMA can work on unstructured grids. Development of the NEPTUNE interface began fairly recently, so is not as mature as some of the other projects. However, good progress is being made, and it is now possible to compute a 3Dvar increment with the static B from BUMP. Current efforts are focused on building up the set of observations interfaced with NEPTUNE.

LFRic

LFRic (named for Lewis Fry Richardson) is the next generation and exa-scale ready forecast model in development at the Met Office in the UK. LFRic uses the GungHo dynamical core (Staniforth et al. 2014), which uses mixed finite element finite volume solvers. Since LFRic itself is still in early development, the interfacing efforts

have been using an aqua planet version of the model.

The main focus of the past six months has been on putting together a prototype 4DEnVar system running across multiple nodes, which was a key deliverable earlier this year. In the process of developing a 4DEnVar system, other systems have all been developed and tested, to include 3DVar, 3DEnVar, and 3D-FGAT. The LFRic interface can use in-situ surface, radiosonde, and aircraft observations, as well as radiances simulated using the Radiative Transfer for TOVS (RTTOV) operator. Implementing the RTTOV operator into the UFO was a key development and deliverable, and this will enable comparison of RTTOV with the CRTM for all models.

Summary and Future Work

The JEDI development effort began almost from scratch around two years ago. Nevertheless, tremendous progress has been made towards producing realistic analysis states for a number of next generation atmospheric and marine forecast models. This has required a significant parallel development effort to provide data assimilation algorithms, a generic B matrix operator, observation processing, and observation modeling. Around 15 models are now interfaced to JEDI in some form, and many different data assimilation algorithms, with a range of these models, are being run on a daily basis. So far, native grid 3DVar, 3DEnVar, 3D-FGAT, 4DEnVar, and 4DVar have all been implemented for operational forecast models. Using the UFO, in-situ, radiance, bending angle, satellite wind, surface pressure, and marine observations are all being assimilated from several different platforms. Having a generic B matrix operator has been crucial in making quick progress, and all models are successfully making use of the BUMP software for producing static and ensemble B matrix operators on their native grids.

Over the coming months, the focus of the atmospheric groups is on refining the static B matrix operator and testing it within configurations with operational-like horizontal grid spacing. Currently, the atmospheric models being interfaced are using a univariate approach to the static B operator. In most operational systems, a multivariate approach is used, typically through balance operators. In order to achieve this on the native grids, the model requires development of capability not existing in current operational systems. Fortunately, this issue is not applicable to the SOCA interface, where it has been possible to implement a multivariate B matrix operator with refactoring of existing techniques. With a more realistic B matrix, and as more observation operators become available through the UFO, the atmospheric modeling groups will be looking to perform longer and higher resolution cycled experiments.

The first versions of the SOCA, MOM6, and CICE interfaces were delivered to NOAA-EMC in September 2019. EMC's current planning involves replacing the variational component of the Global Ocean Data Assimilation System (GODAS) with SOCA in early 2020. Between now and passing the first SOCA release to EMC, efforts will be directed towards increased resolution in cycled experiments, improving efficiency in the system, and examining the viability of using a hybrid 3DVar approach harnessing the existing LETKF ensemble in GODAS.

Authors
Daniel Holdaway (JCSDA), Guillaume Vernieres (JCSDA), Marek Wlasak (Met Office), Sarah King (NRL)

References

Adcroft, A and Hallberg, R. 2006. "On methods for solving the oceanic equations of motion in generalized vertical coordinates." *Ocean Modelling* 224-233.

Cotter, C. .J and Thuburn, J. 2014. "A finite element exterior calculus framework for the rotating shallow-water equations." *Journal of Computational Physics* (Journal of Computational Physics).

Giraldo, F. X., Kelly, J. F., and Constantinescu, E. 2013. "Implicit explicit formulations of a three-dimensional nonhydrostatic unified model of the atmosphere (NUMA)." *SIAM J. Sci. Comput.* B1162–B1194.

Lin, S.-J. 2004. "A "Vertically Lagrangian" Finite-Volume Dynamical Core for Global Models." *Mon. Wea. Rev.* 2293-2307.

McMillen, J., 2019: Prediction and Data Assimilation for Cloud (PANDA-C). *JCSDA Quarterly, 65*, Fall 2019

Staniforth, A, Melvin, T, Wood, N. 2014. "GungHo! A new dynamical core for the Unified Model." ECMWF. https://www.ecmwf.int/sites/default/files/elibrary/2014/12389-gungho-new-dynamical-core-unified-model.pdf.

Walters, D. N., E. C. Hunke, C. M. Harris, A. E. West, J. K. Ridley, A. B. Keen, H. T. Hewitt, and J. G. L. Rae. 2015. "Development of the Global Sea Ice 6.0 CICE configuration for the Met Office Global Coupled model." *Geoscientific Model Development* 2221–2230.

Observations in the Joint Effort for Data Assimilation Integration (JEDI) - Unified Forward Operator (UFO) and Interface for Observation Data Access (IODA)

Introduction

The Joint Effort for Data assimilation Integration (JEDI) system is designed to be generic, flexible, and computationally efficient. JEDI decouples observation operators, observation filtering and data assimilation algorithms from state-space operations and packages them as separate components (classes) that can be re-used for a variety of applications in a wide range of environments from operational forecasting to academic research (Trémolet, 2020, this issue). JEDI is implemented primarily in C++ (and can link to Fortran code) using a generic and object-oriented approach. This article highlights the two key components of the JEDI framework related to observations: the Unified Forward Operator (UFO) and the Interface for Observation Data Access (IODA).

The Unified Forward Operator (UFO)

The UFO code links with the Object Oriented Prediction System (OOPS) and implements classes related to the observation operators, which express the computation of a simulated observation given a known model state. All operators are written so that they can be used generically, that is with any model. This separation of the observation operator code from the model code reduces the amount of time required to introduce a new model into JEDI. Additionally, the UFO code implements generic classes that manipulate the results of observation operators for bias correction and quality control, which of course is critical in any data assimilation system.

Observation Operators

A broad introduction to how observation operators fit into JEDI is discussed earlier in this newsletter (Trémolet, 2020, this issue). As discussed, observation operators are split into two parts ***(see Figure 1)*** to allow for different models (with different State class implementations) to use the same observation operator.

The first part of the full observation operator, named getValues in OOPS, is model (geometry)-dependent and is implemented with the model interfaces. It extracts the values of model variables (GeoVaLs) at desired observation locations (Locations) and typically performs interpolation of requested model variables to the observation locations. This part of the observation operator, therefore, is model-aware, and this component isolates model details from the remainder of the operator code.

The second part of the full observation operator is model-independent and computes the Observation Operator from the input GeoVaLs. This part is implemented in UFO and is called ObsOperator in OOPS. ObsOperators can be rather simple, like computing wind speed from directional u- and v-wind components or vertically interpolating measurements in a profile. They can also be quite complex, such as when simulating radiometer brightness temperatures at several instrument channels.

Both GeoVaLs and Locations classes are implemented in UFO since they are meant to be used by all model interfaces. At present, the Locations class considers latitude, longitude, and time. It will soon be extended to more sophisticated modeling geometries with the ability to consider footprint locations and slanted profiles.

UFO already implements many ObsOperators, including:

- Brightness temperature and radiance simulations using both the Community Radiative Transfer Model (CRTM; Han et al., 2006) and the Radiative Transfer for TOVS (RTTOV; Saunders et al., 2018),
- Vertical interpolation in various specified coordinate systems (can be used with radiosonde, aircraft, satwind observations),

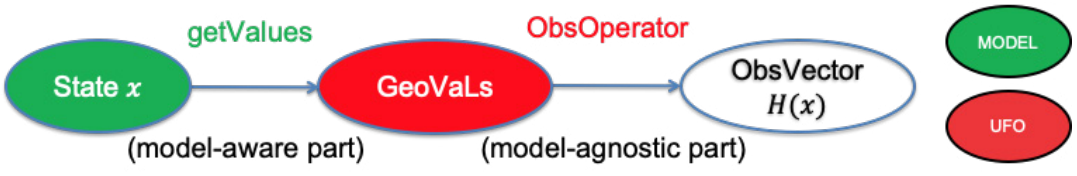

Figure 1. *Observation operator in JEDI.*

- Calculations of aerosol optical depth (by invoking CRTM),
- Calculations of sea ice thickness and sea ice fraction, and
- Calculations of GNSS-RO bending angles (1D and 2D; Shao et al., 2019) using both NCEP bending angle simulations and an interface to the Radio Occultation Processing Package (ROPP).

In addition to computing the result of the observation operator H(x), ObsOperators may also return additional information, like Jacobians and ancillary diagnostic data produced by the operator, that are not directly used in the assimilation algorithm but could be useful for quality control (see next section), bias correction, or scientific investigations.

Quality Control and ObsFilters

Quality control in JEDI is handled through observation filters (class ObsFilter in OOPS). Filters can change quality control flags (i.e., to reject or retain observations) and observation error variances (e.g., one might wish to increase observation error variances to decrease the observation weight in the analysis instead of rejecting observations altogether). The filters can use information from:

- Observation metadata and observed values,
- GeoVaLs (model fields at observation locations),
- Results of observation operators (H(x)),
- Optional diagnostics (ObsDiagnostics) output from the observation operator, and
- Results of functions of some combination of the above variables. These functions (implementations of ObsFunction in UFO) are pre-compiled but may have user-customizable parameters.

The UFO ObsFilters are written to be generic. Generic filters do not depend on specific observation types and are configured through a configuration file. They may be applied either before or after an observation operator. This customizability allows the UFO to encompass much of the pre- and post-processing work performed in existing codes running in operational centers.

Some examples of generic filters include:

- A Background check that looks for differences between observation value and model simulated value (y – H(x)) and compares it to an absolute and/or relative threshold (a relative threshold is a threshold multiplied by observation error),
- A Domain check that verifies if some metadata, GeoVaLs, or observation operator diagnostics are within specified limits, and
- Thinning of observations based on specified criteria.

All filters can be configured to perform various actions depending on filter results. To illustrate the use of generic filters, consider several examples using the same generic filter (Domain Check).

The first example (below) can be used with GNSSRO data and rejects all observations with observation altitudes outside of a range of zero to 30 km:

```
Filter: Domain Check
  where:
  - variable: {name: altitude@MetaData}
    minvalue: 0
    maxvalue: 30000
  action:
    name: reject
```

The second example (below) can be used in the ocean data assimilation and rejects all observations for which the model's sea area fraction is smaller than 0.5:

```
Filter: Domain Check
  where:
  - variable: {name: sea_area_fraction@GeoVaLs}
    minvalue: 0.5
  action:
    name: reject
```

The third example (below) uses result of function that computes wind speed from the wind components to inflate observation error by a factor of two for all observations for which wind speed is higher than 40:

```
Filter: Domain Check
  where:
  - variable: {name: Velocity@ObsFunction}
    maxvalue: 40.0
  action:
    name: inflate error
    inflation: 2.0
```

The generic observation filters, combined with the very flexible control by yaml configuration files is a very powerful tool. No new code is required to apply a filter to different observation types, which also applies to new observations. Scientists can focus on the scientific aspects of their work, reducing the amount of time necessary to assimilate new observations.

Bias Correction

Another essential component of data assimilation is bias correction. Here, bias represents systematic differences between actual sensor measurements and a model's simulated results. This can occur from a variety of processes. Biases are typically represented by a predictor model involving properties of the observed atmospheric column (such as the integrated lapse rate), model air-mass components (such as 1000-300 hPa thickness), and instrument state (e.g., its field of view). In particular, bias correction is an important and necessary step when assimilating radiance data. In general, this bias is associated with a given instrument and frequency band (channel) and varies in space and time. It depends in large part on atmospheric conditions at the time of observation, although other sources

of radiometric bias include the radiative transfer model itself, surface emissivity and orography, and instrument antenna characteristics.

Many bias correction schemes have been developed and implemented in NWP systems. These schemes apply an offset to minimize the systematic differences between sensor measurements and the results of forward operators. In literature, several approaches are employed, including (and by no means limited to) slope-intercept correction (using an offline linear regression of measurements against simulations), histogram adjustment (re-centering the histogram of the differences between simulations and measurements to make it centered around zero), variational approaches and multi-layer convolutional neural networks (Dee et al., 2009; Zhu et al., 2014; Tao et al., 2016).

To give developers and users flexible and configurable bias correction scheme interfaces, a factory design pattern is adopted, which provides approach to code for interface rather than implementation. At the time of this writing, the predictor model of linear regression formulation and the predictors used by GSI have been implemented and tested in UFO. However, the UFO code's generic bias correction interface allows a user to specify what predictors they want to use and what predictor model they want to apply in a configurable YAML file, increasing the flexibility of the scheme.

Like the rest of the UFO, the design of the interfaces for bias correction, and the flexibility for selecting a bias correction scheme through a yaml configuration file, will facilitate future scientific developments. All the data available to observation filters, from observation data, GeoVaLs, observation operators output and diagnostics, and functions of those, can be used for bias correction. New bias corrections schemes can be implemented more easily. New or existing schemes and predictors can easily be applied to new observation types without any coding.

The Interface for Observation Data Access (IODA)

Overview

The objective of the IODA is to provide uniform access to observation data across the whole forecasting chain from observation pre-processing to data assimilation to diagnostics.

The initial effort in the project has focused on the in-memory data access, which is a generic implementation of the OOPS ObsSpace class (Trémolet, 2020). It is used extensively with the observation operators, quality control, and bias correction in UFO and contributes to the genericity of those components. The second direction of effort is the definition of the IODA file format. It will enable more generic exchange of information throughout the forecasting suite and caters for efficient I/O to populate or save data from the in-memory IODA structure. Finally, a third direction of effort will be the long-term storage and retrieval of observation data and diagnostics in an organized structure (a data store). The common file format and organized data store will enable exchanges of data and scientific evaluations and comparisons.

In-memory Observation Handling

As its name indicates, IODA is primarily

an interface that isolates the scientific code, for example, in the observation operators, from the underlying data structures holding the data. The logical structure users should have in mind when using IODA is described below. The actual data storage might differ if a different organization has advantages in terms of efficiency or maintenance.

At the center of the scheme are tables which hold the data. Tables are two-dimensional. The columns represent variables (e.g., temperature, humidity, brightness temperature) and the rows represent the locations. Associated with each axis of the data tables are metadata tables. The location metadata hold the values describing each location and which are appropriate for each observation type (e.g., latitude, longitude, and scan angle). The variable metadata holds values associated with each variable (row) in the data tables, again appropriate per observation type, such as variable names or channel frequencies. Finally, the data table is replicated dynamically as necessary in a third dimension to hold different kinds of data, such as the observations themselves, the observation error, quality control flags, or simulated observations H(x) at various stages of the data assimilation process ***(see Figure 2)***.

The metadata tables along the two axes are fully generic and might contain very different information depending on the observation type. Additional functionality, for example, to access data by station identifier, by flight, or by sonde ascent, are being added to the interface. The IODA structure can contain data of different types, for example, observation values are real numbers while quality control flags are integers. This capability can be used in the future to store more complex information, for example, observation operator Jacobians. This also applies to the metadata tables.

The IODA interface provides methods to

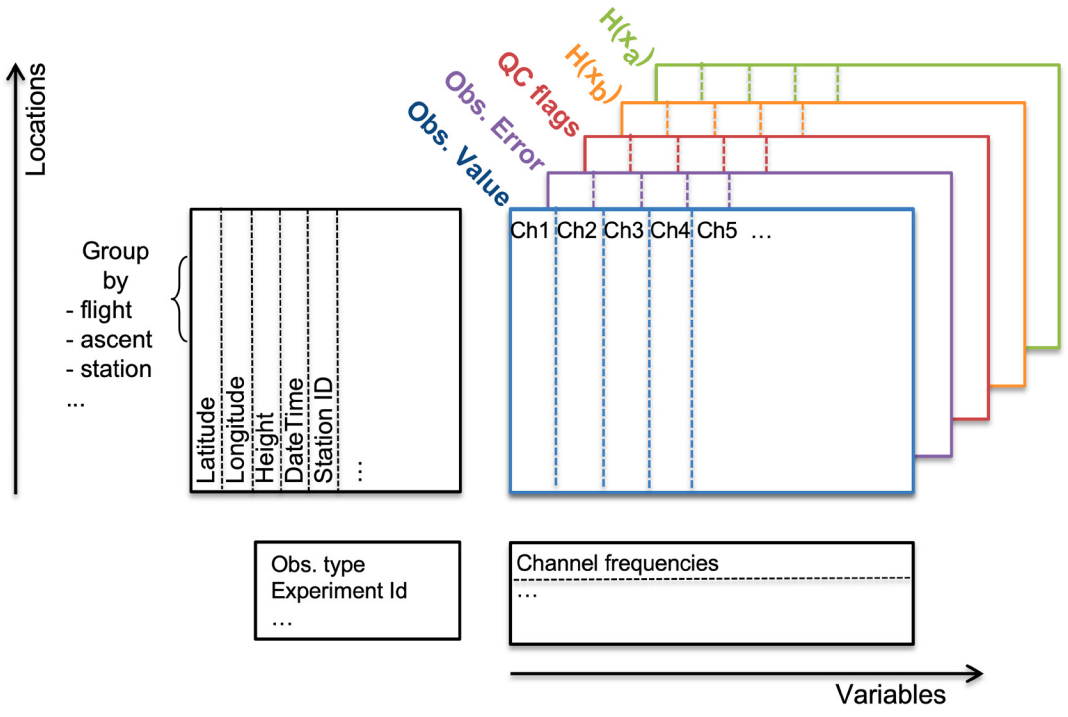

Figure 2. *Schematic representation of the IODA data organization.*

query information about the data and to access data to and from the in-memory storage. An interface is provided for both Fortran and C++ (below). The interfaces are generic enough to cover all use cases by observation operators, quality control, and bias correction for all observation types. However, they also give flexibility for optimizations in the implementation. For example, the distribution of observations on distributed memory architectures should be handled at the implementation level and be transparent to the user. The IODA interface provides the separation of concerns between the scientific aspects of the code and the efficiency and software engineering aspects.

IODA interface example in C++:

```
std::vector<double> lats;
obsspace.get_db("MetaData", "latitude", lats);

std::vector<float> viewing_angle;
obsspace.get_db("MetaData", "sensor_view_angle", viewing_angle);
```

IODA interface example in Fortran:

```
nlocs = obsspace_get_nlocs(self%obsdb)
allocate(obsValue(nlocs))
call obsspace_get_db(self%obsdb, "ObsValue", "bending_angle", obsValue)

allocate(obselev(nlocs))
call obsspace_get_db(self%obsdb, "MetaData", "station_elevation", obselev)
```

IODA Files

Operational centers receive observation data in many different formats (e.g., BUFR, GRIB, netCDF, and HDF5) and often generate several intermediate formats during observation pre-processing, data assimilation, and for diagnostics purposes. IODA offers a single file format for all observations and for use throughout the forecasting chain. Typically, one file will contain observations for one data assimilation cycle, but the format can accommodate other cases, for example, for diagnostics over longer time periods. As the IODA files will be used in operational applications and full resolution research experiments, I/O efficiency, including for distributed applications, is a primary concern in the design. Currently, two file formats are being evaluated for use in IODA, namely netCDF4 and ODB. Preliminary investigations show that both file formats could be suitable; more advanced evaluations are being conducted at the time of writing to inform a decision (that could be taken by the time this article is published).

To handle the various file formats in which observation data are presented, IODA provides a set of conversion programs that translate those formats into the IODA file format. Currently observation data can be converted from BUFR, netCDF4, ODB, and a specialized binary format for marine observations. This is an active area of development as numerous new observation types are being brought on board in JEDI.

The IODA code base is split into two primary components ***(Figure 3)***, "ioda," which implements the memory storage level and "ioda-converters," which supplies a means for converting the various file formats from the observation data providers into the IODA file format. Also depicted in Figure 3 is the typical data flow (black arrows) for the conversion and consumption of observation data by the JEDI system.

Observation Data Store

In addition to handling the observations during a data assimilation cycle, it is important to store observations and additional information for longer term. This is for analyzing scientific results, for repeating experiments, or for future use in reanalysis. Given the collaborative nature of JCSDA and the many partners involved, observations will have to be accessible from a variety of platforms, from laptops and desktops to cloud computing instances to dedicated HPC systems. The design and implementation of such a data store will start in 2020.

Ongoing Efforts

Both IODA and UFO underwent heavy development in 2019. In particular, the JCSDA hosted one workshop and two code sprints in February, April, and August. In February 2019, the U.S. Navy hosted a requirements-gathering workshop for IODA in Monterey, California (Herbener, 2019). This was a highly productive and collaborative effort with representation from eight JCSDA partner organizations. The workshop resulted in a solid set of requirements that are driving the development of the IODA subsystem.

The code sprint in April 2019 took place in Boulder, CO, and was focused on the development of the capability to run 3D-Var cycling experiments with the marine model SOCA. Over the course of two weeks, converters were written for the ingest of marine observations including sea ice thickness and fraction, sea surface temperature and height, altimetry and salinity, along with the corresponding forward operators. The code sprint resulted

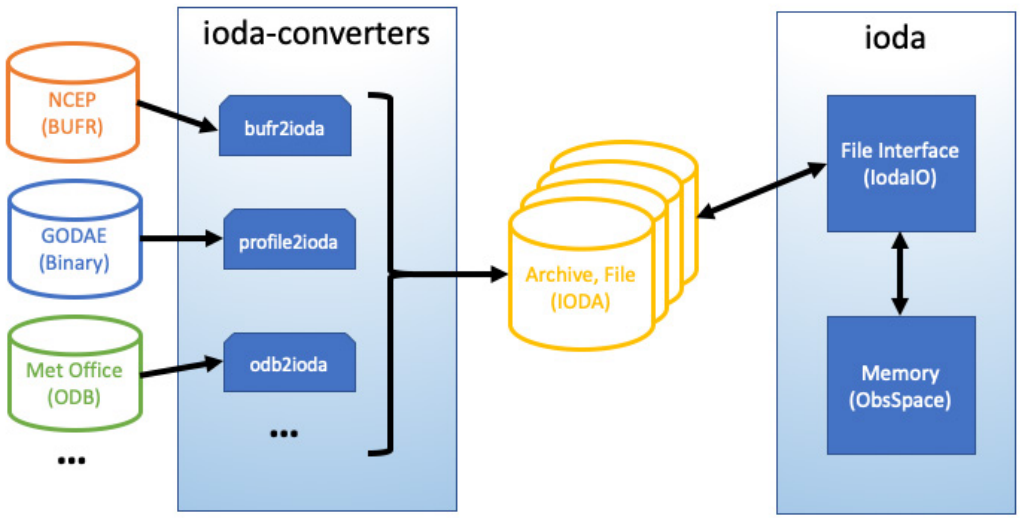

Figure 3. *IODA components: "ioda-converters" holds programs for translating various file formats (BUFR, netCDF4, specialized binary, ODB, etc.) into the IODA file format. "Ioda" contains the implementation of the in-memory storage.*

in the demonstration of a functional full month cycling using the new observations, with the SOCA model.

The second code sprint (August 2019) also took place in Boulder, CO, and lasted two weeks. This effort was focused on developing UFO QC filtering functions for a large set of microwave and infrared instruments, as well as the introduction of radar observations into JEDI. The necessary converters and observation operators were written for handling the new observation types, along with the development of generic filtering functions.

Collaborative work between all JCSDA partners has been very successful. We would like to use JEDI to reproduce the behavior and results of multiple operational systems in 2020. To that end, collaborative development will continue outside code sprints, and code sprints will be organized when beneficial. Quality control procedures will be extended to cover the full range of observations used by JCSDA partners in operational systems, for example, GSI. The observation operators, quality control, and variational bias correction will be validated by comparison with operational systems.

In the longer term, all these operators will be developed further to improve on the current operational systems, taking advantage of new technologies, such as machine learning and artificial intelligence. Collectively, the JCSDA partners have the capacity to develop the largest collection of high-quality observation operators and associated procedures. UFO and IODA provide the tools to share and further improve them.

Authors

Ryan Honeyager (JCSDA), Stephen Herbener (JCSDA), Xin Zhang (JCSDA), Anna Shlyaeva (JCSDA), and Yannick Trémolet (JCSDA).

References

Dee, D. P. and Uppala, S., 2009: Variational bias correction of satellite radiance data in the ERA-Interim reanalysis. *Q.J.R. Meteorol*. Soc., 135: 1830-1841, https://doi.org/10.1002/qj.493.

Han, Y., Van Delst, P., Liu, Q., Weng, F., Yan, B., Treadon, R., and Derber, J., 2006: JCSDA Community Radiative Transfer Model (CRTM) - Version 1. *NOAA Technical Report NESDIS*, 122, https://repository.library.noaa.gov/view/noaa/1157/noaa_1157_DS1.pdf.

Herbener, S. R., 2019: "2019 IODA Workshop Summary." *JCSDA Quarterly*, 63, 23-25, https://doi.org/10.25923/c23x-ac34.

Saunders, R., Hocking, J., Turner, E., Rayer, P., Rundle, D., Brunel, P., Vidot, J., Roquet, P., Matricardi, M., Geer, A., Bormann, N., and Lupu, C., 2018: An update on the RTTOV fast radiative transfer model (currently at version 12). *Geosci. Model Dev.*, 11, 2717-2737, https://doi.org/10.5194/gmd-11-2717-2018.

Shao, H., Vandenberghe, F., Zhang, H., Ruston, B., Healy, S., Cucurull, L., 2019: "Development of GNSS-RO Operators for JEDI/UFO." *JCSDA Quarterly*, 62, Winter 2019, https://doi.org/10.25923/w2dh-ep66.

Tao, Y., X. Gao, K. Hsu, S. Sorooshian, and A. Ihler, 2016: A Deep Neural Network Modeling Framework to Reduce Bias in Satellite Precipitation Products. *J. Hydrometeor.*, 17, 931–945, https://doi.org/10.1175/JHM-D-15-0075.1.

Trémolet, Y., 2020: Joint Effort for Data assimilation Integration (JEDI) Design and Structure. *JCSDA Quarterly*, 66, Winter 2020 (this issue).

Zhu, Y., Derber, J., Collard, A., Dee, D., Treadon, R., Gayno, G. and Jung, J. A., 2014: Enhanced radiance bias correction in the National Centers for Environmental Prediction's Gridpoint Statistical Interpolation data assimilation system. *Q.J.R. Meteorol. Soc.*, 140, 1479-1492, https://doi.org/10.1002/qj.2233.

The Joint Effort for Data Assimilation Integration (JEDI) Infrastructure

Overview

Like the JCSDA itself, the Joint Effort for Data assimilation Integration (JEDI) project literally begins with the idea of collaboration. As part of a Joint Center, we serve a diverse community of forecasters, researchers, academics, and policy makers; and, as part of a Joint Effort, our software development team includes the JEDI core staff and in-kind support from Joint Center for Satellite Data Assimilation (JCSDA) partners, as well as external collaborators who are not only users but also developers, making valuable contributions to the JEDI code base that serve to enhance their own applications as well as others. Potential JEDI users range from operational forecasters to university researchers to graduate students studying data assimilation. Potential applications range from coupled Earth system and Numerical Weather Prediction (NWP) models to idealized "toy" models designed to probe the fundamental physics of atmospheric and oceanic flows.

To support this diverse community, we need collaborative workflows and software tools that leverage contributions from distributed developers while promoting a unified vision. We also need infrastructure that will allow users and developers to build and run JEDI on a range of computational platforms, from laptops and workstations to local Linux clusters, to cloud computing instances, to high-performance computing (HPC) systems at national research facilities and operational centers. These are the challenges, and this article describes how we are meeting those challenges.

Modern challenges require modern strategies. Over the past several decades, the concept of agile software development has revolutionized the technology industry. Different tech companies may define this concept somewhat differently, depending on whether or not they adopt a specific agile strategy, such as Scrum or Kanban, but all agile approaches share some common principles. In particular, agile software development is often contrasted with a waterfall approach in which software requirements are laid out at the beginning of the development process and the finished product is delivered to users months or even years later. Agile software development principles followed by the JEDI project include:

1. **Innovation, technical excellence, and *people over process***
   Software development should leverage the skills of talented, motivated individuals and the latest technical advances. Team members are given the freedom to explore innovative solutions and share responsibility for the quality of the finished product.
2. **Continuous delivery of functional software**
   Working software is the primary measure of progress. Innovations should be coded, tested, and delivered to the users promptly and frequently, without waiting for the product to be "done" (an agile system is never "done").
3. **Responsive to users and other stakeholders**
   Requirements change. As the project proceeds, users may desire new functionality or encounter unforeseen circumstances. The software must adapt accordingly.
4. **Flexibility and simplicity**
   No software is truly "future-proof," but an agile software development approach appreciates that demands on the product are likely to change and embraces this expectation. For JEDI this means accommodations for new models, new satellite missions, new observations, new algorithms, new workflows, new computing platforms, and for data that continually grows ever bigger. An aspect of this that is often cast as a separate principle is maintaining simplicity in both the code and the development process. Simple, straightforward, modular code facilitates maintenance and enhancement.

We achieve these ideals by means of a number of software tools, most notably the git version control system, the web-based GitHub platform for managing, archiving, and distributing code, and the ZenHub project management software. After describing how we implement these tools for agile software development in the next section, we then discuss how we promote the implementation of JEDI software across computing platforms through the use of software containers, machine images, and environment modules. Next, we address how to promote innovation, excellence, and continuous delivery (items 1 and 2 above) through automated testing and how we further serve our community (items 3 and 4) through versatile NWP workflows.

A common thread throughout is cloud computing. The cloud provides an unprecedented opportunity to efficiently distribute code, data, computing services, and computing resources to our community of JEDI developers and users,

and JCSDA has made a commitment to exploit this opportunity. As the technical capabilities of cloud-hosted computing and storage resources continue to improve and price continues to drop, these platforms also provide a viable alternative to HPC facilities for many JEDI users who may have limited access to national HPC centers or local clusters.

The cloud also provides an exceptional opportunity for training new JEDI users and developers. JCSDA holds two JEDI Academies every year, at varying locations, where participants are introduced to the structure of the code, the working practices that guide code development, and the process by which they can build and run applications. Each afternoon, Academy participants are given hands-on experience building and running JEDI on cloud computing instances that are pre-provisioned with an appropriate environment. Such instances can be created and destroyed as needed and may also be used for online tutorials in the future.

Software Development and Distribution

The **git** version control system provides a powerful way to organize code into repositories, track changes, and define code branches that isolate new developments until they are completed, tested, and ready to be merged into the primary code base. It runs on a user's local laptop, workstation, cluster, or HPC platform; wherever code development and testing take place. Code changes are made available to other developers and users by means of the web-based **GitHub** platform, which is closely integrated with git. Developers can push and pull code to and from GitHub multiple times per day with a few simple git commands. GitHub represents each git repository as a separate web site, with dropdown menus to access different branches and additional tools to view differences between branches. In order to limit the size of the GitHub code repositories, large data files used for testing are stored remotely on the cloud and access either directly or via GitHub's Large File Service (LFS).

Git and GitHub are powerful tools, but they require a strategy in order to exploit them for optimal benefit. The strategy used at JEDI is one that has come to be known as **git-flow** (Driessen 2010). In the git-flow paradigm ***(Figure 1)***, each repository has only two permanent branches. "Permanent" meaning the branches themselves exist indefinitely while the code they contain evolves. The first is the **master** branch, which is reserved for public releases that, in the case of JEDI, we plan to make open-source. The second permanent branch is the **develop** branch. As its name suggests, this branch is for code in development, but it is important to emphasize that all code in the develop branch must be functional and well tested.

The vast majority of the code development occurs in **feature** branches that split off from the develop branch. As they work on a feature branch, each developer makes code changes that enhance functionality or address known issues, and they also write new tests for the code they have added. As they work, they continually push their code to GitHub where other developers can access it from anywhere, at any time. In an agile framework, it is important to keep the feature branches focused on a specific development that will, ideally, take no more than a few weeks to implement. The lifetime

of some feature branches may even be a few days or less.

When a feature branch is mature and well tested, the developer announces that it is ready to be merged into the develop branch by issuing what is called a **pull request** on GitHub. All pull requests are subject to **automated testing** and **code reviews** by other members of the JEDI team. Details about the automated testing framework are discussed in the following section. Limiting the scope of each feature branch allows reviewers to more thoroughly assess the proposed changes. Often reviewers suggest changes to the feature branch that are discussed through the GitHub interface and then implemented by the original author of the feature branch or by any other member of the team. When at least two (often more) reviewers agree that the feature branch is ready, then a member of the JEDI team (with administrative privileges for that repository) merges the changes into the develop branch. The feature branch is then deleted and the developer moves on to another issue.

Other code branches admitted by the git-flow paradigm include **bugfix** branches that fix known problems in the develop branch and **hotfix** branches that fix known problems in the master branch (for example, patches to releases). There are also **release** branches that are used to make final refinements prior to a release, such as adding documentation or adding tests. Like feature branches, bugfix, hotfix, and release branches are temporary and are deleted after they are merged into the develop and/or master branches. And, like feature branches, they are intended to be limited in scope to facilitate code reviews.

The git-flow paradigm is extremely effective at promoting the agile software development principles discussed in the opening section. The coexistence of multiple feature branches together with release, bugfix, hotfix, develop, and master branches allows the JEDI team to pursue innovation while simultaneously and continually delivering functional software to the user (agile principles 1-2). To further promote these and the remaining agile principles (1-4), the ZenHub project management tool is used.

ZenHub integrates smoothly with GitHub but is a separate, third-party, web-based application. It enables users to create project boards similar to those used in agile workflows like scrum or kanban. Each project board is linked to one or more GitHub repositories and is populated by tasks, referred to as issues. These may include the

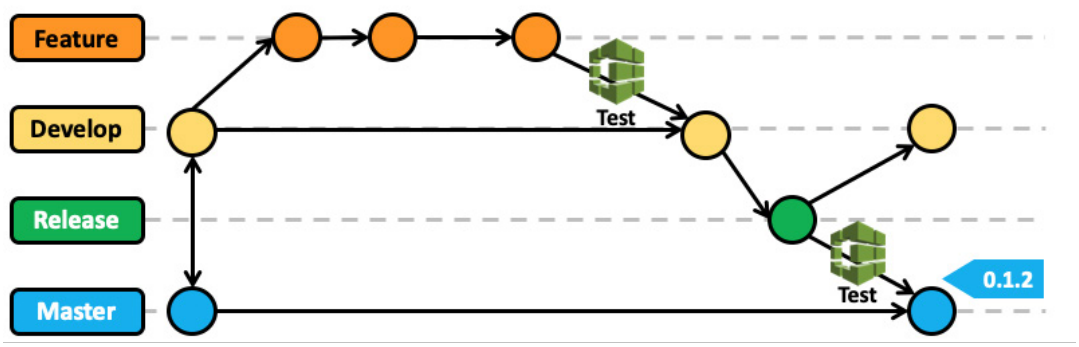

Figure 1. *The git-flow branching paradigm followed by JEDI. Several code branches are highlighted as discussed in the text. All merges to the develop and master branches must be done through GitHub after passing code reviews and automated testing, as indicated by the AWS CodBuild icon. All merges to the master branch are tagged with a release number (here 0.1.2).*

development of new features or bug fixes or discussion threads. The issues are organized into a series of columns that include the project **backlog** (a *to do* list), tasks that are **in progress**, tasks that are under review or being discussed (review/QA), and tasks that have been completed or closed. There is also an icebox column for low-priority tasks that do not require immediate attention.

Issues may be prioritized within each column and assigned to one or more team members. Those following the issue are informed when its status has changed. For example, followers are notified by email when another team member comments on the issue or move it to the in progress column. This is an effective way to maximize collaboration and minimize time lost through duplication.

ZenHub also has many other tools for organizing issues into milestones (used for code sprints), epics (used for project accounting and long-term planning), and releases (used for defining software releases and other deliverables). Zenhub also has a number of reporting tools, allowing managers and team members to generate essential agile standbys, such as burndown charts, velocity tracking, and release reports.

Documentation on how to build and run the JEDI system is provided through online user manuals generated with the sphinx document generator and published through ReadtheDocs.com. More detailed low-level code descriptions and diagrams are also generated by means of the Doxygen software package.

Portability

The diverse user community described in the opening section poses an implementation challenge: how can we support those who wish to run JEDI on laptops, workstations, clusters, and supercomputers? To best serve our users, JEDI needs to be portable.

An important component of our portability strategy is the relatively new technology of software containers. Though the general concept of software containers has been around for decades, it has only matured in recent years, following the release of Docker in 2013. Docker greatly facilitated the creation and distribution of containers, which established its current position as the most popular container provider. But Docker is an enterprise product that is not well suited for scientific applications. In particular, security vulnerabilities make Docker impractical for most HPC systems. Singularity was developed to address these limitations and to thereby promote the use of containers for scientific workflows. For those who do not have access to Singularity, we also support a third container option called Charliecloud. Charliecloud can be installed and run by any user on any viable linux system without the need for any privileged system access (Singularity requires administrative privileges to install, but not to run).

Briefly, the idea behind a software container is to encapsulate a computing environment, often distributed as a single file, in such a way that lets each user re-create that same environment on their own computer, whether it be a laptop, a cloud instance, or an HPC system. In the case of JEDI, this computing environment includes a number of third-party software libraries, such as NetCDF, CMake, and LAPACK, all built with a specific set of compilers (such as gnu, clang, or intel), and MPI libraries (such

as OpenMPI, mpich, or Intel MPI) --- in short, everything that is needed to build (if necessary) and run JEDI. So, for example, a user can just download the JEDI Singularity container, "enter" that container with a single command, and then proceed to run JEDI applications.

Two types of containers are supported. The first are development containers that include the compilers and compiled dependencies but do not include the JEDI code itself. Once inside the container, user/developers can pull the JEDI code from GitHub, modify it as needed, and then proceed to compile and run it. By contrast, application containers are more streamlined. They do not include the compilers (which can make the container files large and unwieldy), but they do contain a compiled, tagged release of the JEDI code, ready to go. Our workflow begins by first generating a JEDI Docker container for each compiler/MPI combination. Then we create the Singularity and Charliecloud containers directly from these Docker containers. An advantage of this workflow is that the Docker development containers are also available for continuous integration testing, as described in the next section.

Though containers can be run on HPC and cloud platforms, it is often beneficial to install the JEDI dependencies directly on these systems, exploiting site-specific configurations and optimizations. These are made available to users as environment modules that can be loaded with a single command. The use of environment modules promotes optimal performance and also allows developers to easily switch between compiler/mpi implementations and library versions. We currently maintain JEDI environment modules on selected HPC systems used by JCSDA and its partners. On the Amazon cloud, we provide these environment modules to users through bootable Amazon Machine Images (AMIs). All these environment modules are built using the same set of build scripts used to create the containers, fostering a uniform computing environment for JEDI users and developers. This minimizes problems associated with incompatible software versions or inadequate configuration options.

Continuous Integration

As the software industry favored frequent and small modifications to the code (agile methodology) for the development approach, the need for frequent automated testing became paramount. Most of the Continuous Integration (CI) services, such as Travis-CI or Amazon Web Services (AWS) CodeBuild, use webhooks that are triggered by various events on the repository hosting site. The automated testing system has the capability to perform various tasks based on the webhook event type. For example, with new changes to a repository, a certain set of tests can be triggered and run.

The automated testing framework in JEDI is designed to build the application and run the tests with every new pull request (to the develop and master branches) and every push to an existing pull request on the GitHub repository. The status of the test is shown on the pull request page for the developers and reviewers. Passing all tests ensures the developers that the new feature is compatible with all the JEDI components and can be added to the repository. With automated testing, any error or incompatibility in the new scripts can be caught at the early stages of the development and can help to run the development pipeline more efficiently.

Automated testing can help to make the review process shorter and to add new features to the repository quicker.

Building the required libraries and environment can be time-consuming especially for JEDI applications that depend on large libraries. By using a prebuilt environment inside a Docker container, we can reduce the build time and allocate most of the resources to building and testing the JEDI repositories. Separate docker containers can be built with different compilers and library versions. These containers are used in parallel to test the software across multiple platforms in the shortest amount of time. Currently, we use two containers for automated testing purposes. The first is based on GNU compilers and is hosted on DockerHub and is accessible for all users. The second is based on Intel compilers and, due to licensing restrictions, is hosted on the AWS Elastic Container Registry (ECR) and is only available to JCSDA AWS users.

Currently, Travis-CI and AWS CodeBuild automated testing services are implemented in the JEDI core repositories. Both of these services provide customers with cloud computing resources to run tests automatically. With every new commit to the repository, source code is downloaded from the GitHub repository onto the automated testing server. Docker containers are used to provide all the necessary libraries and packages required to build and run the application. The next stage is to build and run the tests inside the Docker environment and on the automated testing server and to report the test status on the pull request GitHub page.

Travis-CI computational resources are limited and suitable for less computationally expensive but more frequent tests compared to AWS CodeBuild. With AWS CodeBuild, three different instance sizes are available that users can choose based on their computational needs. For each JEDI core repository (i.e. oops, saber, ioda, and ufo) and for the FV3-jedi repository, two CodeBuild projects are set with GNU-based and Intel-based containers that run in parallel.

Another feature that is currently being added to the testing framework is a multi-tier testing capability. As part of this feature, tests in each repository will be classified in different tiers based on their computational cost. For example, tests that use high-resolution reference files are classified as more expensive tests or high-tier compared to small unit tests that are classified as low-tier tests. Low-tier tests will run more often than high-tier tests to speed up the testing process and reduce computational costs. When building and testing low-tier tests, large reference files used in high-tier testing will not be downloaded to speed up the testing and reduce the data bandwidth usage. High-tier tests will set to run periodically (i.e., daily, weekly, or monthly) to ensure that every aspect of the code is being tested.

After building JEDI and running the tests, CodeCov is used to create a report on the test coverage. The test coverage report highlights the sections of the code that are not fully tested so developers can focus on writing tests for these sections. CodeCov also calculates how much new changes (with pull requests) change the test coverage and reports it to the GitHub pull request page ***(Figure 2)***.

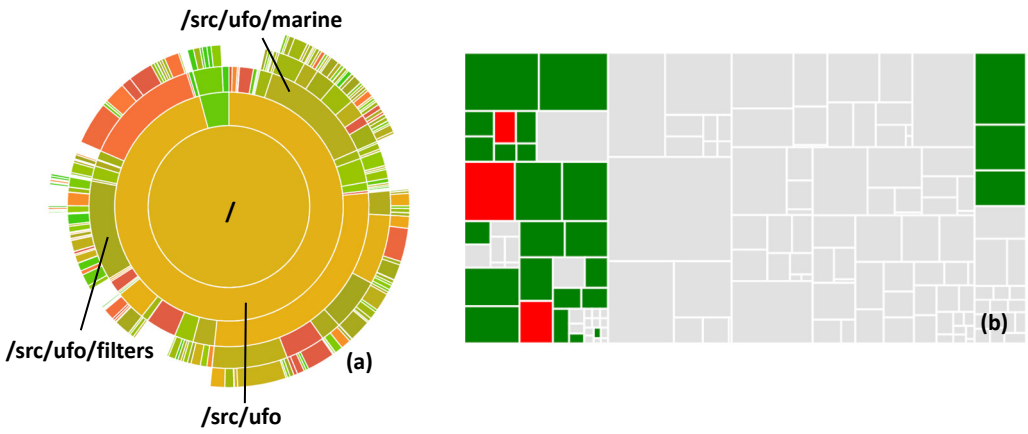

Figure 2. *a) Coverage sunburst plot is an interactive plot that shows the test coverage in each directory of the repository (ufo illustrated here). b) Changes in test coverage with each pull request is illustrated in an interactive plot as well. Each rectangle represents a file in the repository and how changes in this pull request increased (green) or decreased (red) the test coverage in these files.*

Workflow

In the JEDI project, genericity is a core principle present in every aspect of library and infrastructure design. The JEDI system consists of generic programs that interface Earth-system models together with collections of DA algorithms and observation operators. Likewise, the JEDI container infrastructure allows these programs to be generically ported and tested across compilers, MPI distributions, operating systems, and computer architectures. Together, these features of the JEDI system define a combinatorial matrix of models, algorithms, operating systems, partner institutions, and hardware resources. The purpose of the workflow management component of the JEDI project is to provide a set of composable generic applications covering the space implied by these multiple axes of genericity. Using modern software design techniques, our goal is to provide users with a uniform, simple, and powerful interface for designing, modifying, scheduling, and monitoring data-centric workflows composed of JEDI executables. The scalability of the JEDI system and infrastructure require the workflow management software also to scale with the application resource requirements, encompassing everything from debugging DA algorithms on a laptop, to running multi-node experiments in the cloud, to managing operational-grade cycling forecast systems on HPC resources.

The JEDI workflow system, JEDI-Rapids, consists of two parts: (1) a set of generic, composable *applications* corresponding to JEDI executables and their associated data, and (2) a generic *workflow-generator* allowing customizable generation of graphical workflow structures that combine individual JEDI applications into full analysis toolchains. As a workflow-generation system, the JEDI-Rapids system is designed to programmatically produce concrete workflow descriptions for a range of production-quality workflow management software engines including ecFlow, Cylc, and Apache Airflow ***(Figure 3)***. The workflow management engines are then responsible for scheduling and monitoring the execution of applications and executables.

In order to present a uniform interface, configuration of the JEDI executables, the Python applications that control them, and the connection of applications

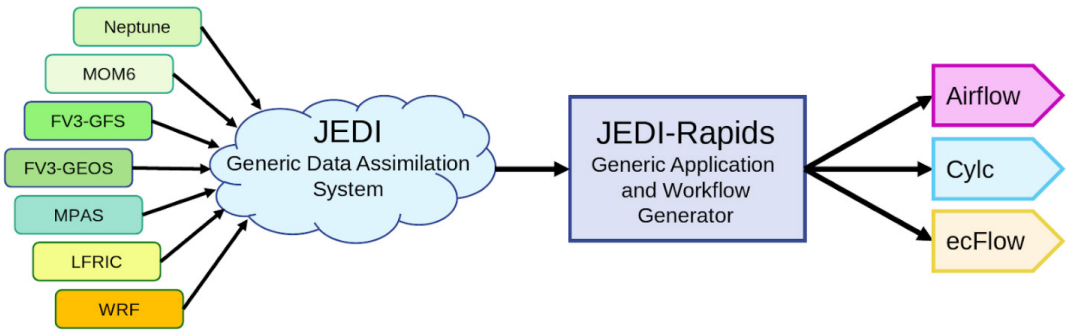

Figure 3. *The JEDI-Rapids system is a generic workflow generator, interfacing JEDI applications with a range of different operational-grade workflow managers.*

into larger workflow specifications, are each accomplished with YAML-syntax configuration files using the Jinja templating engine. The combination of YAML and Jinja is simultaneously powerful, simple, and easily editable, allowing the user to quickly reconfigure workflow descriptions. A user can change model parameters, DA algorithms, covariance models, observation operators, and observation QC filtering algorithms, as well as the entire workflow graph structure, all without writing any shell scripts, editing any code, or recompiling any packages. Future work will add a web-based graphical interface layer over-top of the YAML configuration to make automation and monitoring of common workflows even more intuitive and flexible.

Finally, the JEDI-Rapids system aims to improve the robustness, repeatability, and comparability of analysis products for Earth-system modeling applications. Key focuses of the JEDI workflow system are: (1) *experimental reproducibility*, the ability to re-run and compare results on wildly different systems; and (2) *data provenance*, the ability to precisely trace the origin of each analysis product. Executable portability across systems is directly supported by the design of the JEDI infrastructure including the containers, environment modules, and CMake build system. The JEDI-Rapids system is designed to enable the wider goal of repeatability of entire experimental analysis pipelines across disparate systems. The object-oriented and functional programming constructs of the Python language are key to this ability, enabling the Python-based workflow applications to dynamically reconfigure interface components allowing them to adapt to different execution environments in ways not possible for systems based on ordinary shell scripts. As a result, the more broadly analysis toolchains become portable, the more important it becomes to also accurately document the origin of analysis products. The JEDI-Rapids system is also designed to track data-provenance through the use of metadata stored as ordinary YAML-syntax files. The provenance metadata associates each analysis product to the particular system, execution environment, software versions, input data products, and application configuration settings used in its generation. In conjunction, these features allow JEDI applications and workflows to be easily ported with the JEDI-Rapids system to match the infrastructure and requirements of partner institutions, while simultaneously allowing analysis products to be compared and checked for correctness.

Authors
Maryam Abdi-Oskouei, Mark Miesch, and Mark Olah (JCSDA)

References
Driessen, V. 2010, "A Successful Git Branching Model," https://nvie.com/posts/a-successful-git-branching-model/

PEOPLE

Welcome Dr. Dick Dee

Dick Dee joined the JCSDA in October 2019, as a Senior Research Scientist. His role at the Joint Center will be to provide leadership in the area of observations.

Dick is originally from the Netherlands but spent most of his career outside his home country. He has a doctorate in Applied Mathematics from the Courant Institute of Mathematical Sciences at New York University, where he first learned about data assimilation and its application to numerical weather forecasting. Early in his career, he worked as a math professor at the Pontifícia Universidade Católica in Rio de Janeiro, Brazil, as a research professor at New York University, and, subsequently, as a research scientist at Delft Hydraulics in the Netherlands. He then returned to the U.S., joining the newly formed Data Assimilation Office (now GMAO) at NASA. During this time, he made several contributions to data assimilation science on topics, such as adaptive Kalman filters, covariance estimation, model bias correction, and observation quality control. As a visiting scientist at ECMWF in 2003, he implemented the variational bias correction (VarBC) component of the Integrated Forecast System (IFS). In 2005, he returned to ECMWF to work on reanalysis (including ERA-Interim) and led the ERA-CLIM projects involving satellite data rescue, coupled data assimilation, and production of century-long coupled climate re-analyses. In 2014, he became Deputy Head of the new Copernicus Climate Change Service at ECMWF, overseeing activities related to production of climate data records, climate reanalysis, seasonal forecasting, and development of a cloud-based Climate Data Store.

Dick considers himself a very lucky man, both in his personal and professional life, who always ends up in the right place with the right people. He is happy to be back in the U.S., where he always feels very much at ease. He is thrilled to join a young, groundbreaking, dynamic team that is hell-bent on changing the NWP world order.

Introducing Dr. Wei Han

Dr. Wei Han joined the New and Improved Observation (NIO) team at UCAR/JCSDA in October 2019. His primary responsibility and focus are the evaluation and assimilation of current and future geostationary hyper-spectral Infra-Red sounders. He physically works at the Space Science and Engineering Center (SSEC) at the University of Wisconsin-Madison as a UCAR project scientist, collaborating with SSEC and JCSDA team members, and also engages with Joint Effort for Data assimilation Integration (JEDI) at the JCSDA associated with this type of data. In the last two years, he has focused on the assimilation of the first geostationary hyper-spectral IR sounder(the Geostationary Interferometric Infrared Sounder (GIIRS)) on-board FY-4A. GIIRS temperature sounding channels have been operationally assimilated in Chinese Meteorological Administration's Numerical Weather Prediction (NWP) system GRAPES since December 25, 2018, using 4D-Var.

Wei is from China, where he earned his PhD and MS degrees in Meteorology. His research is focused on the development of operational variational data assimilation system in the last 15 years with focus on the bias correction of satellite data. Dr. Wei Han has developed Constrained Bias Correction (CBC) and Constrained VarBC (CVarBC) schemes for satellite radiances assimilation to constrain the size of the bias correction using uncertainty information from calibration and radiative transfer model, in order to avoid the drift to model bias. The CVarBC was successfully implemented in the ECMWF IFS for satellite radiance data assimilation by providing further constraints using potential available information, such as constraints on the size of the bias correction and innovative bias correction metrics using uncertainty estimation from calibration and radiative transfer. This has been studied in the full ECMWF global 4D-Var system using data from microwave sounders, which are sensitive to stratospheric temperature and ozone-sensitive infrared radiances from IASI, AIRS, and CrIS. The constrained VarBC of AMSU-A stratospheric sounding channels reduces the biases in the stratosphere and improves the medium range forecasts in the stratosphere and troposphere. The CVarBC has been activated in ECMWF IFS 45R1 since June 5, 2018. Wei also investigated the assimilation of IASI ozone channels in 2009 as an NWP-SAF visiting scientist at ECMWF. On November 15, 2011, ECMWF implemented the operational assimilation of ozone-sensitive infrared radiances from AIRS, IASI, and HIRS, which represents a major milestone in exploiting infrared sounders and analyzing ozone.

Apart from science, Wei loves exploring, traveling, and running. He has completed eight full marathons since 2008.

Say Hello to Dr. Andy Fox

Andy Fox joined the JCSDA in Boulder, CO, in July to lead our efforts in land surface model data assimilation. He has conducted research in this field for the last two decades through a variety of projects focused on the roles of land surface processes in the coupled Earth System.

Originally from Manchester, UK, Andy has degrees in Geography and Hydrology from the University of Oxford, the University of Colorado - Boulder, and a PhD from the University of Cambridge. Since, he has worked as a researcher at a number of universities in the UK and in the US. He has strong connections with NCAR where he was a visiting scientist for a number of years developing a data assimilation system for the Community Land Model. Outside of academia, he gained project management and system engineering experience whilst working in the data products group at the National Ecological Observatory Network (NEON). Andy is well known for his work on the terrestrial carbon cycle and is currently the chair of the Science Leadership Group of the inter-agency North American Carbon Program. He is particularly excited about what the next generation of satellite observations will bring to our understanding of the land surface and how these new observations will be used in advanced, coupled data assimilation systems.

Andy has called Boulder home for many years and enjoys life by the mountains skiing in the winter and climbing and trail running in the summertime.

Welcome Dr. Nan Chen

Dr. Nan Chen joined the JCSDA in September 2019, as an Associate Scientist with the JEDI NIO team. Primarily Dr. Chen is working on getting new observations into the JEDI system. He works closely with the JEDI core team and has extensive experience in various fields, including satellite data application, algorithm development, and radiative transfer simulations.

Nan graduated in May 2016, from the Light and Life Lab at Stevens Institute of Technology studying radiative transfer and satellite remote sensing. His work during that time included radiative transfer modeling of atmosphere, cloud, snow, land, and ocean in SW and IR regions, as well as its applications. He then developed a machine learning-based cloud mask and snow parameter retrieval algorithm based exclusively on a radiative transfer simulated dataset for the cryosphere mission of the Japan Aerospace Exploration Agency (JAXA) GCOM-C1/SGLI project. During and after his PhD study, Nan developed the skills and tools required to achieve successful outcomes in scientific research projects, a discipline well suited to the vision of the JCSDA program.

In his spare time, Nan likes cooking, running, hiking, biking, Ping-Pong, badminton, and playing competitive computer games.

EDITOR'S NOTE

I doubt that any topic associated with the Joint Center for Satellite Data Assimilation has inspired more conversation in our community during the past couple of years than the Joint Effort for Data assimilation Integration (JEDI) Project. Indeed, JEDI likely raises more questions than just about any other aspect of the JCSDA, ranging from the most basic ("What is it, actually?") to the technical ("How is it being built, and does it work yet?) to the playful ("How much effort have you invested in devising so many cute acronyms?) This issue is composed of a five-part, JEDI-focused series of articles; collectively, you will find that they address a great many of these questions (at least the serious ones) and moreover, they may motivate more working DA specialists to draw from and contribute to JEDI in the future.

Yannick Trémolet and Tom Auligné have written an introductory article that explains the need for data assimilation development that is generic, reusable, open, and agile, in order to support a variety of diverse, rapidly-evolving environmental models, as well as a growing and increasingly varied global observation system that include more short-lived satellites. The high-level structure of JEDI (linking generic functions to abstract interfaces and, in turn, to specific modeling system implementations) is described in a second article authored by Trémolet.

In fact, the development of interfaces to numerous model systems of the JCSDA partners is now well underway, and the status of these interfaces within JEDI is summarized in a third article written by Daniel Holdaway, Guillaume Vernieres, Marek Wlasak, and Sarah King. The fourth article, by Ryan Honeyager, Stephen Herbener, Xin Zhang, Anna Shlyaeva, and Yannick Trémolet, describes a pair of JEDI components that are critical to the actual use of observations in JEDI: the Unified Forward Operator (UFO) and the Interface for Observation Access (IODA). The final piece in the package tackles the infrastructure of JEDI (that is, the software development, distribution, and workflow supporting the JEDI objectives of portability and continuous integration) is from Maryam Abdi-Oskouei, Mark Miesch, and Mark Olah.

A number of new colleagues have joined the JCSDA during the last quarter. They are Andy Fox, who will be leading the nascent Land DA Project, Nan Chen, who joins the New and Improved Observations (NIO) project, Dick Dee, Senior Scientist for Observations, and Wei Han, also working with the NIO team specifically bringing his expertise to bear on hyperspectral IR observations from geosynchronous orbits. You can find biographies of each of the colleagues in this issue to learn more about their work and interests.

As I compose this note, another calendar year is drawing to a close. It has been one of marked success and accomplishment for the JCSDA and all the individuals who contribute to it; I hope that you all look back on it with as much satisfaction and pride as I do. I am confident that we will reach even greater heights in 2020.

Jim Yoe

SCIENCE CALENDAR

UPCOMING EVENTS

MEETINGS AND EVENTS SPONSORED BY JCSDA

DATE	LOCATIONS	WEBSITE	TITLE
February 3–5, 2020	Reading, United Kingdom	https://www.ecmwf.int/en/learning/workshops/4th-workshop-assimilating-satellite-cloud-and-precipitation-observations-nwp	Joint Workshop JCSDA & ECMWF
February 24–27, 2020	Monterey, CA	https://www.jcsda.org/events/2020/2/24/4th-jedi-academy	JEDI Academy 4
February 28, 2020	Monterey, CA	https://www.jcsda.org/events/2020/2/28/crtm-training-amp-user-workshop	CRTM Workshop
June 2–4, 2020	Airforce Academy Colorado Springs, CO	jcsda.org/events	18th JCSDA Technical Review Meeting and Science Workshop

MEETINGS OF INTEREST

DATE	LOCATIONS	WEBSITE	TITLE
February 16–21, 2020	San Diego, CA	https://www.agu.org/Ocean-Sciences-Meeting	Ocean Sciences
May 3–8, 2020	Vienna, Austria	https://www.egu2020.eu/	EGU
June 1–5, 2020	Fort Collins, CO	http://www.isac.cnr.it/~ipwg/	IPWG
June 8–12, 2020	Fort Collins, CO	https://www.cira.colostate.edu/conferences/8th-international-symposium-on-data-assimilation/	8th International Symposium on Data Assimilation (ISDA)
July 19–24, 2020	Waikoloa, HI	https://igarss2020.org/	IGARSS
September 28–October 2, 2020	Wurzburg, Germany	https://www.eumetsat.int/website/home/News/ConferencesandEvents/DAT_4635627.html	EUMETSAT Meteorological Satellite Conference 2020
October 18–23, 2020	Banff, Canada	https://www.birs.ca/events/2020/5-day-workshops/20w5166	Mathematical Approaches for Data Assimilation of Atmospheric Constituents and Inverse Modeling
December 7–11, 2020	San Francisco, CA	https://www.agu.org/	AGU
January 10–14, 2021	New Orleans, LA	https://www.ametsoc.org/index.cfm/ams/	AMS

CAREER OPPORTUNITIES

Opportunities in support of JCSDA may be found at https://www.jcsda.org/opportunities as they become available.