The Joint Effort for Data assimilation Integration (JEDI)

From Objectives to Code Design

Joint Center for Satellite Data Assimilation (JCSDA)

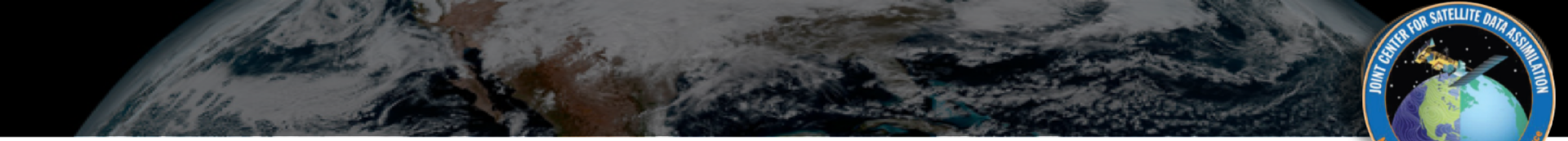
JEDI Academy – 24-27 February 2020

JEDI: Motivations and Objectives

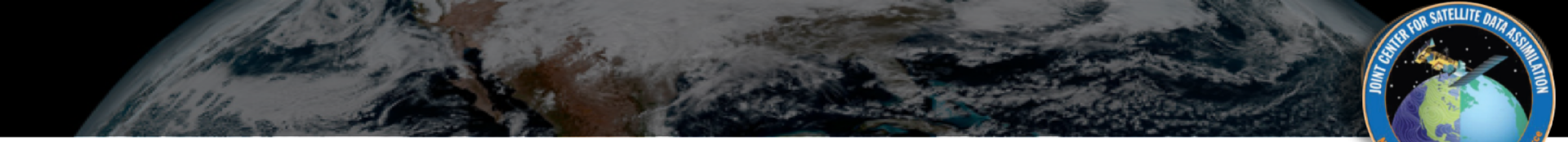
Develop a unified data assimilation system:

- From toy models to Earth system coupled models
- Unified observation (forward) operators (UFO)
- For research and operations (including O2R2O)
- Share as much as possible without imposing one approach (one system, multiple methodologies/configurations)

Objective of this talk: How is such a system designed?

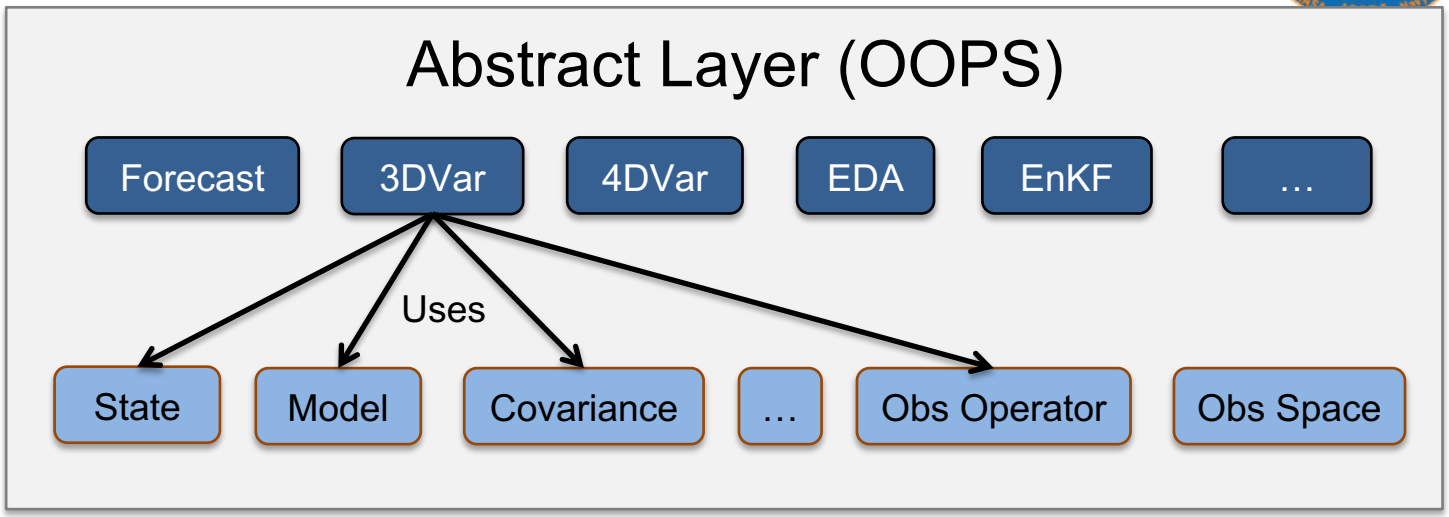


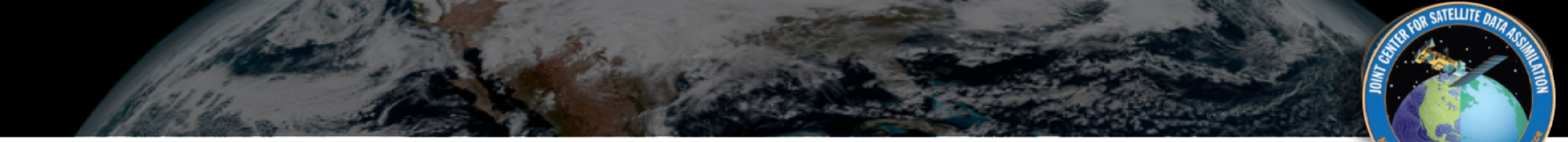
Basic Building Blocks



Generic applications

Interface layer

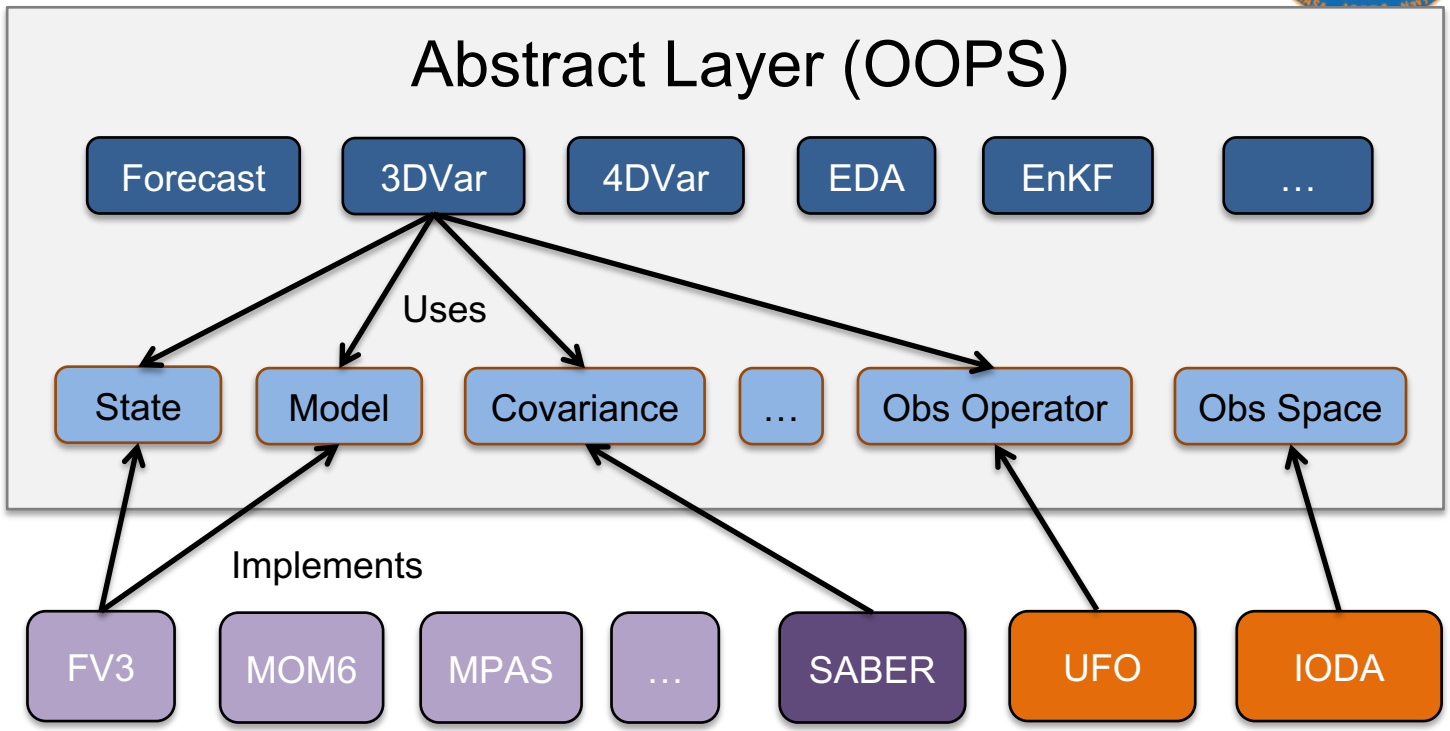


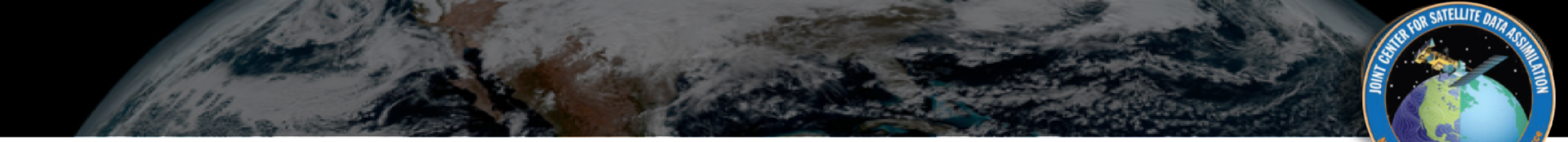


Generic applications

Interface layer

Specific implementations

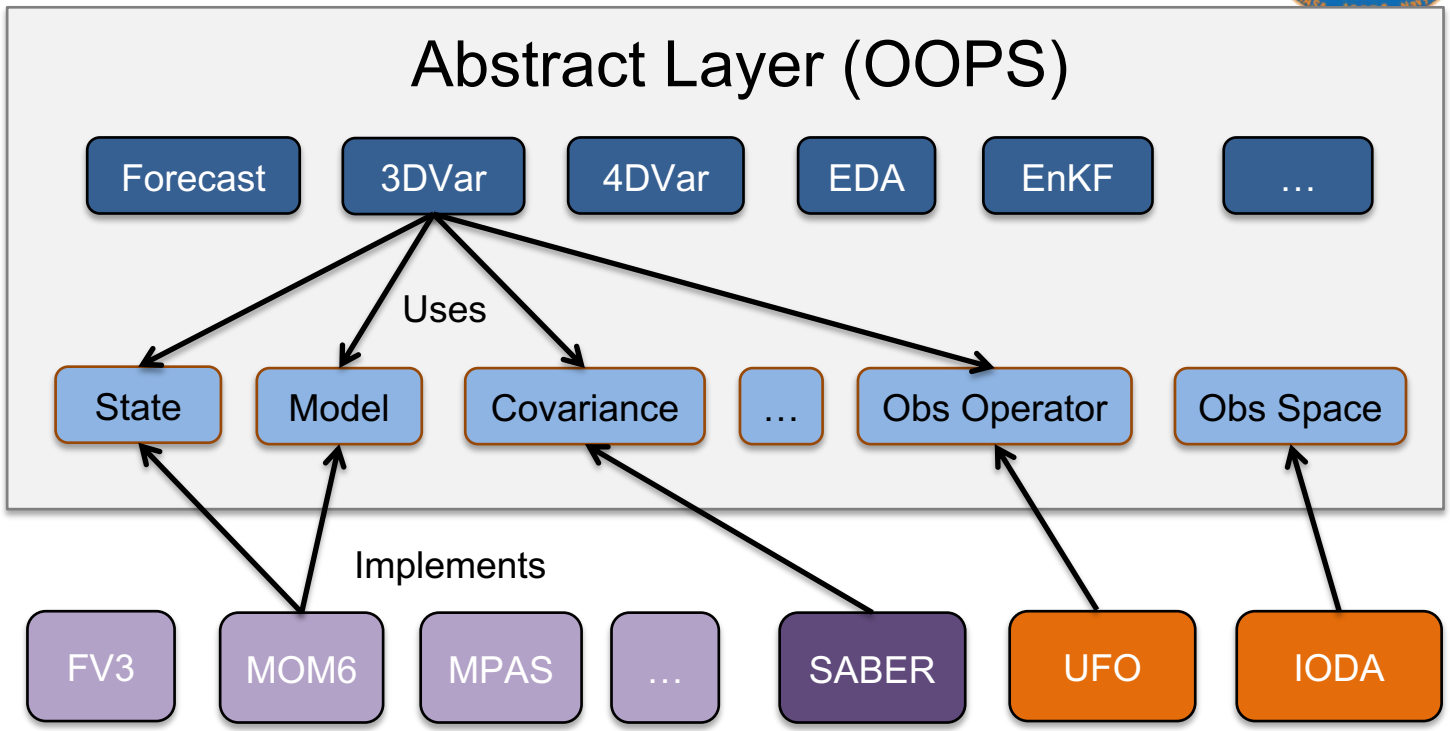


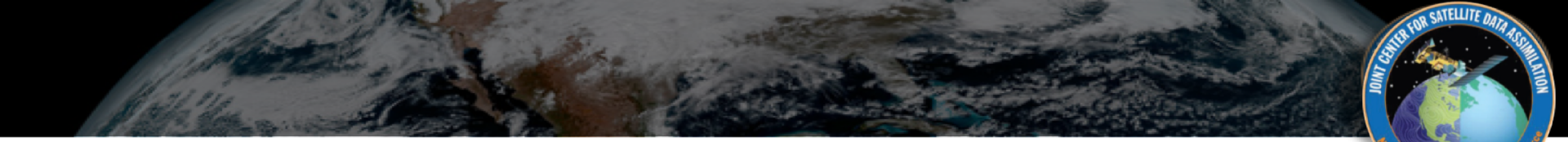


Generic applications

Interface layer

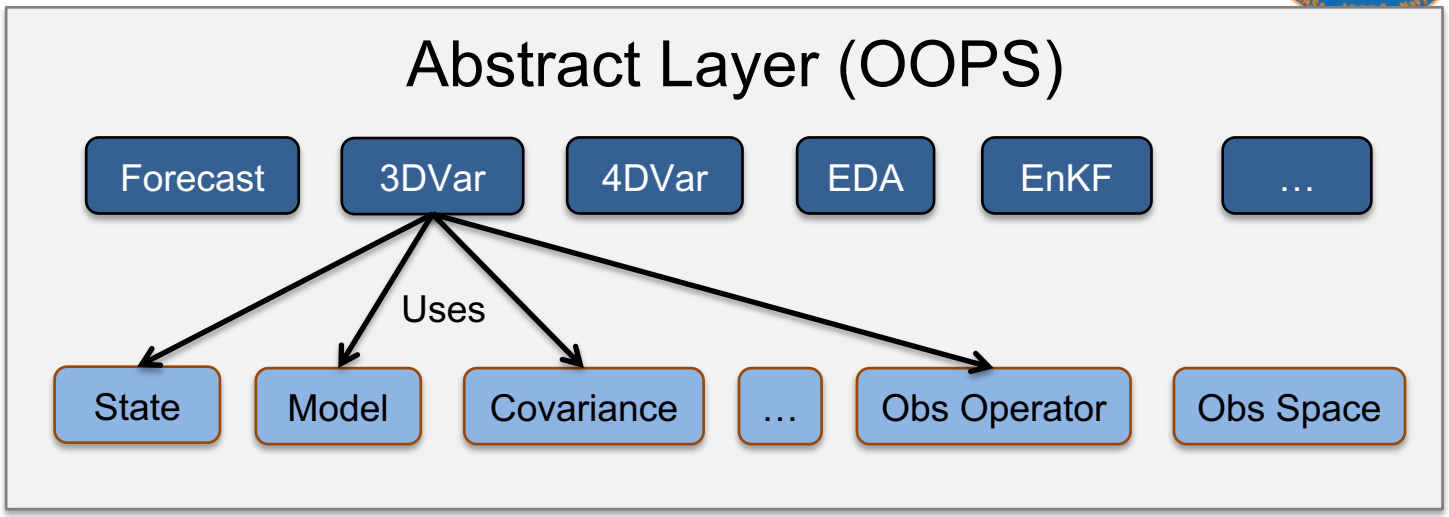
Specific implementations





Generic applications

Interface layer



OOPS Analysis and Design

- All data assimilation methods require the same limited number of entities.
- For future (unknown) developments these entities should be easily reusable.
- These entities are the basic (abstract) classes that define the system.
- No details about how any of the operations are performed, how data is stored or what the model represents: separation of concerns.

Basic building blocks for DA

What is data assimilation?

Data assimilation is finding the best estimate (analysis) of the state of a system given a previous estimate of the state (background) and recent observations of the system.

use variational assimilation and minimize:

$$J(\Delta x) = \frac{1}{2}\Delta x^T \mathbf{B}^{-1}\Delta x + \frac{1}{2}\left(y_o - H(x_b + \Delta x)\right)^T \mathbf{R}^{-1}(y_o - H(x_b + \Delta x))$$

use Kalman filter:

$$\Delta x_a = \mathbf{BH^T}\left(\mathbf{HBH^T} + \mathbf{R}\right)^{-1}(y_o - H(x_b))$$

Model space: State

$$J(\Delta x) = \frac{1}{2}\Delta x^T \mathbf{B}^{-1}\Delta x + \frac{1}{2}\left(y_o - H(x_b + \Delta x)\right)^T \mathbf{R}^{-1}(y_o - H(x_b + \Delta x))$$

$$\Delta x_a = \mathbf{BH^T}\left(\mathbf{HBH^T} + \mathbf{R}\right)^{-1}(y_o - H(x_b))$$

Examples: background, analysis, forecast state

Operations allowed on state:

- Input, output (raw or post-processed).
- Move forward in time (using the model).
- Copy, assign.

From DA point of view no need to know how operations are performed, or how states are represented and stored.

Model space: Increment

$$J(\Delta x) = \frac{1}{2}\Delta x^T \mathbf{B}^{-1} \Delta x + \frac{1}{2}(y_o - H(x_b) - \mathbf{H}\Delta x)^T \mathbf{R}^{-1}(y_o - H(x_b) - \mathbf{H}\Delta x)$$

$$\Delta x_a = \mathbf{B}\mathbf{H}^{\mathbf{T}}\left(\mathbf{H}\mathbf{B}\mathbf{H}^{\mathbf{T}} + \mathbf{R}\right)^{-1}(y_o - H(x_b))$$

Examples: analysis increment, ensemble perturbation

Operations allowed on Increments:

- Basic linear algebra operators,
- Evolve forward in time linearly and backwards with adjoint.
- Compute as difference between states, add to state.

Observations

$$J(\Delta x) = \frac{1}{2}\Delta x^T \mathbf{B}^{-1}\Delta x + \frac{1}{2}\left(y_o - H(x_b + \Delta x)\right)^T \mathbf{R}^{-1}\left(y_o - H(x_b + \Delta x)\right)$$

$$\Delta x_a = \mathbf{BH^T}\left(\mathbf{HBH^T} + \mathbf{R}\right)^{-1}\left(y_o - H(x_b)\right)$$

Examples: observation values, model-simulated observation values.

Operations allowed on Observations:

- Input, output.
- Simulate observation given a state (observation operator).
- Copy, assign.

From DA point of view no need to know how operations are performed, or how observations are represented and stored.

Observations

$$J(\Delta x) = \frac{1}{2}\Delta x^T \mathbf{B}^{-1}\Delta x + \frac{1}{2}\left(y_o - H(x_b + \Delta x)\right)^T \mathbf{R}^{-1}(y_o - H(x_b + \Delta x))$$

$$\Delta x_a = \mathbf{BH^T}\left(\mathbf{HBH^T} + \mathbf{R}\right)^{-1}(y_o - H(x_b))$$

Examples: observation values, model-simulated observation values.

Operations allowed on Observations:

- Input, output.
- Simulate observation given a state (observation operator).
- Copy, assign.

From Observations point of view no need to know how the observations will be used (to compute H(x), in the variational, or Kalman filter assimilation).

Departures

$$J(\Delta x) = \frac{1}{2}\Delta x^T \mathbf{B}^{-1}\Delta x + \frac{1}{2}(y_o - H(x_b) - \mathbf{H}\Delta x)^T \mathbf{R}^{-1}(y_o - H(x_b) - \mathbf{H}\Delta x)$$

Examples: departures, ensemble perturbations in the observation space.

Operations allowed on Departures

- Basic linear algebra operators,
- Compute as difference between observations, add to observations,
- Compute linear variations in observation equivalent as a result of variations of the state (linearized observation operator).
- Output (for diagnostics).

Covariance matrices

$$J(\Delta x) = \frac{1}{2}\Delta x^T \mathbf{B}^{-1}\Delta x + \frac{1}{2}(y_o - H(x_b) - \mathbf{H}\Delta x)^T \mathbf{R}^{-1}(y_o - H(x_b) - \mathbf{H}\Delta x)$$

Examples: background error covariance, observation error covariance (two different interfaces in OOPS)

Operations allowed on Covariance matrices:

- Setup
- Multiply by matrix (and possibly its inverse)

Operators

$$J(x) = \frac{1}{2}(x_0 - x_b)^T \mathbf{B}^{-1}(x_0 - x_b) + \frac{1}{2}\sum_{i=0}^{n}\left(H\left(M_{o\to i}(x_0)\right) - y_i\right)^T \mathbf{R}^{-1}\left(H\left(M_{o\to i}(x_0)\right) - y_i\right)$$

Model operator and its linearized counterpart: $M, \mathbf{M}, \mathbf{M^T}$.

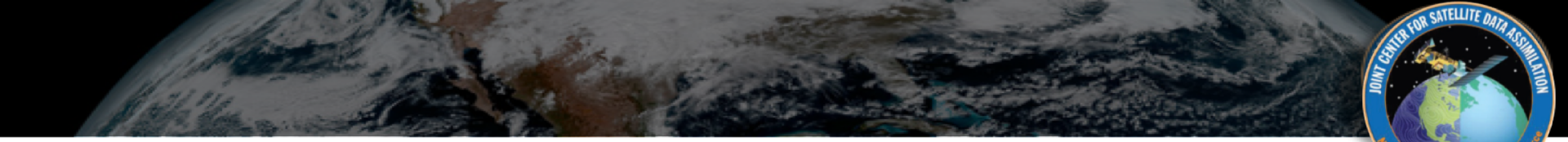
Observation operator and its linearized counterpart: $H, \mathbf{H}, \mathbf{H^T}$

OOPS Analysis and Design

- All data assimilation methods require the same limited number of entities.
- For future (unknown) developments these entities should be easily reusable.
- These entities are the basic (abstract) classes that define the system.
- No details about how any of the operations are performed, how data is stored or what the model represents: separation of concerns.

OOPS Analysis and Design

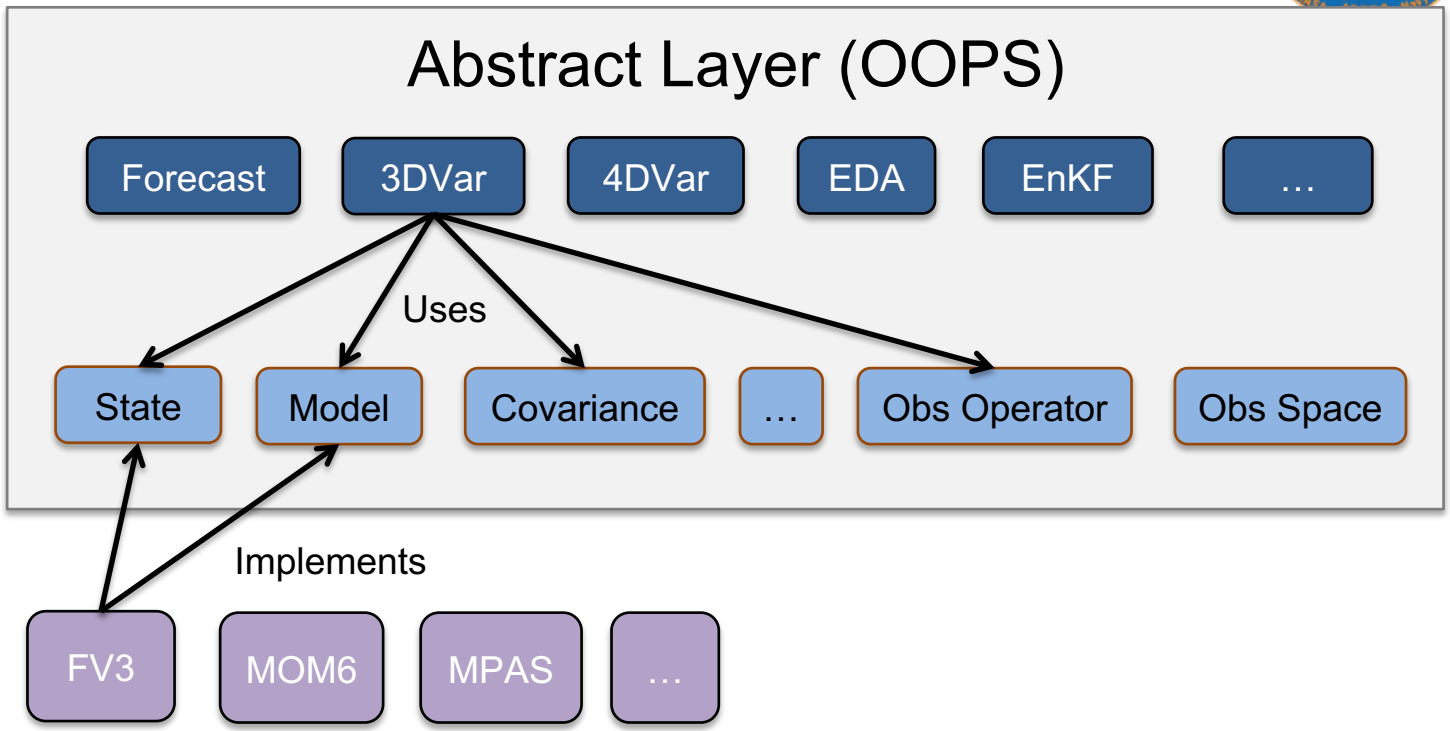
- OOPS is independent of the underlying model and physical system.
- Flexibility (including yet unknown future development) requires that this goes both ways.
- The components do not know about the high level algorithm being run:
  - All actions driven by the top level code,
  - All data, input and output, passed by arguments.
- **Interfaces** must be general enough to cater for all cases, and detailed enough to be able to perform the required actions.



Generic applications

Interface layer

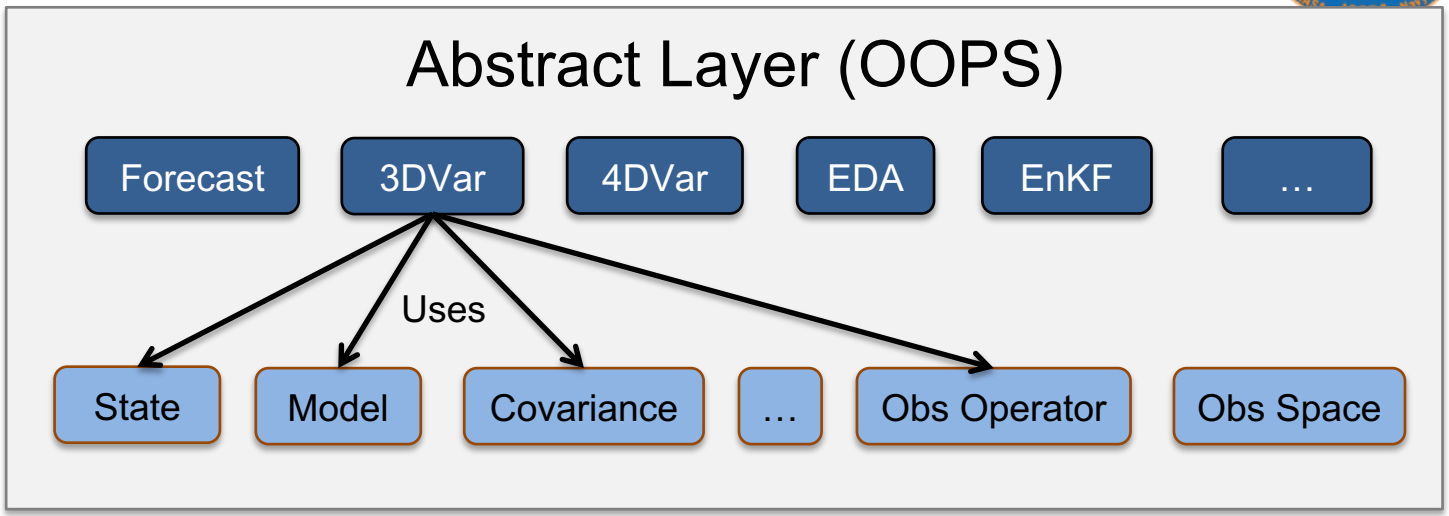
Specific implementations



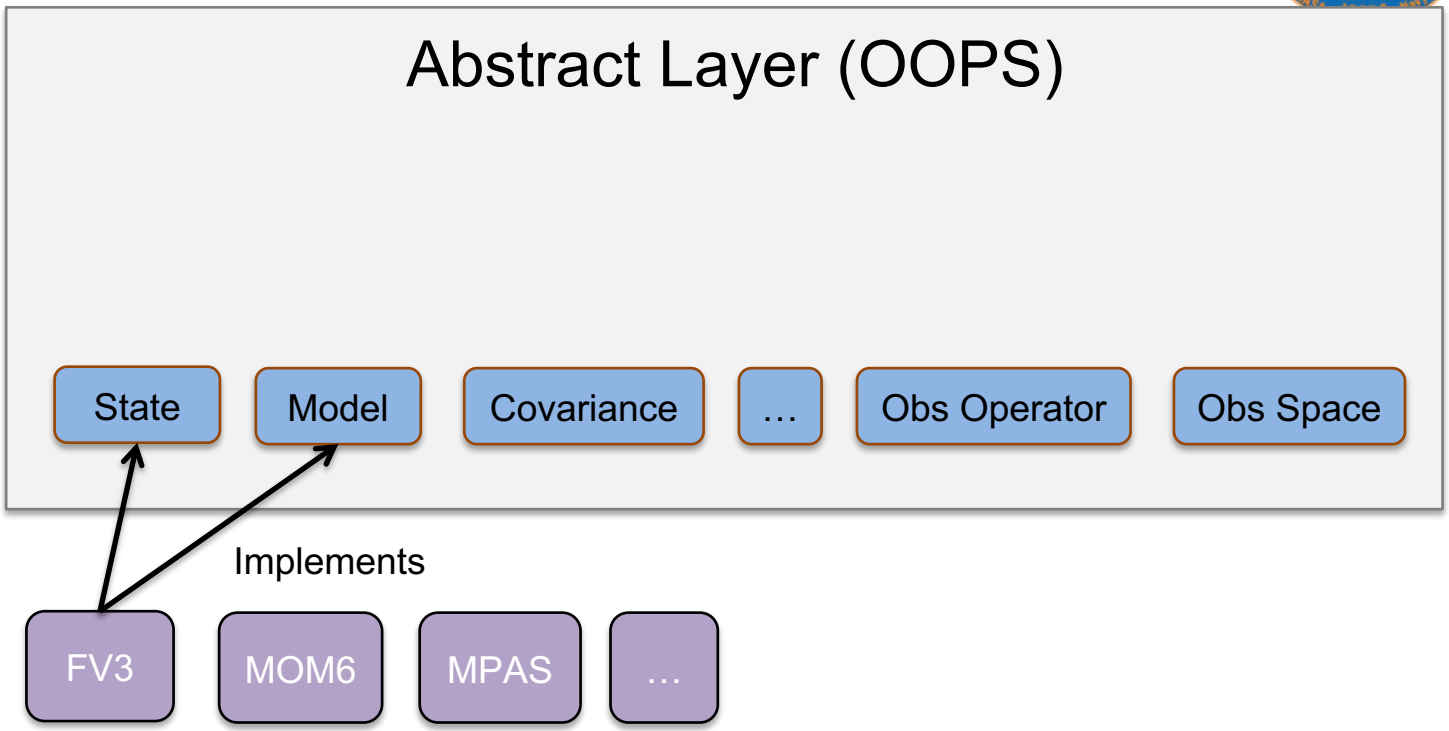
Separation of concerns

Generic applications

Interface layer



Separation of concerns



Interface classes

All model or observation specific classes are wrapped into an interface class

- Using the C++ templates this is not strictly necessary but provides a convenient place to group all interfaces and make them visible

Each interface class contains a (smart) pointer to the actual implementation in the “model trait” and passes calls down

The interface layer is also used to instrument the code

- Timing statistics
- Trace execution

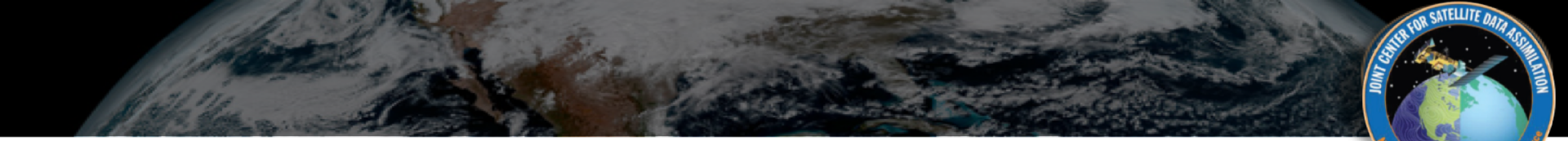
Interface classes are all in `oops/src/oops/interface`

oops directory structure

oops/src/oops/	assimilation	DA classes (minimizer, cost functions, etc)
	base	base classes and classes build up on interface classes (state ensemble, observer, etc)
	generic	implementations that can be shared by different models/obs (diagonal obs errors, BUMP background error covariances)
	interface	interface classes (building blocks from previous slides, need to be implemented)
	parallel	files relevant to mpi communications
	runs	applications (Variational, HofX, EDA, etc)
	util	utilities (datetime, timers, etc)

oops directory structure

oops/src/oops/	assimilation	DA classes (minimizer, cost functions, etc)
	base	base classes and classes build up on interface classes (state ensemble, observer, etc)
	generic	implementations that can be shared by different models/obs (diagonal obs errors, BUMP background error covariances)
	interface	interface classes (building blocks from previous slides, need to be implemented)
	parallel	files relevant to mpi communications
	runs	applications (Variational, HofX, EDA, etc)
	util	utilities (datetime, timers, etc)



Data Assimilation Algorithms

Data assimilation algorithms

- Variational:
  - 3DVar
  - 4DVar
  - weak-constraint 4DVar
  - 4DEnsVar
- EDA (ensemble of variational data assimilations)
- Local Ensemble Transform Kalman Filter (work in progress)

Weak constraint 4DVar

$$J(x) = J_o + J_b + J_c$$

$$J_b = \frac{1}{2}(x_0 - x_b)^T \mathbf{B}^{-1}(x_0 - x_b)$$

$$J_o = \frac{1}{2}\sum_{i=0}^{n}\left(H\left(M_{o\to i}(x_0)\right) - y_i\right)^T \mathbf{R}_i^{-1}\left(H\left(M_{o\to i}(x_0)\right) - y_i\right)$$

$$J_c = \frac{1}{2}\sum_{i=0}^{n}\left(x_i - M_i(x_{i-1})\right)^T \mathbf{Q}_i^{-1}\left(x_i - M_i(x_{i-1})\right)$$

Involves running model for J_o and J_c steps

Cost Function Design

$$J(x) = J_o + J_b + J_c$$

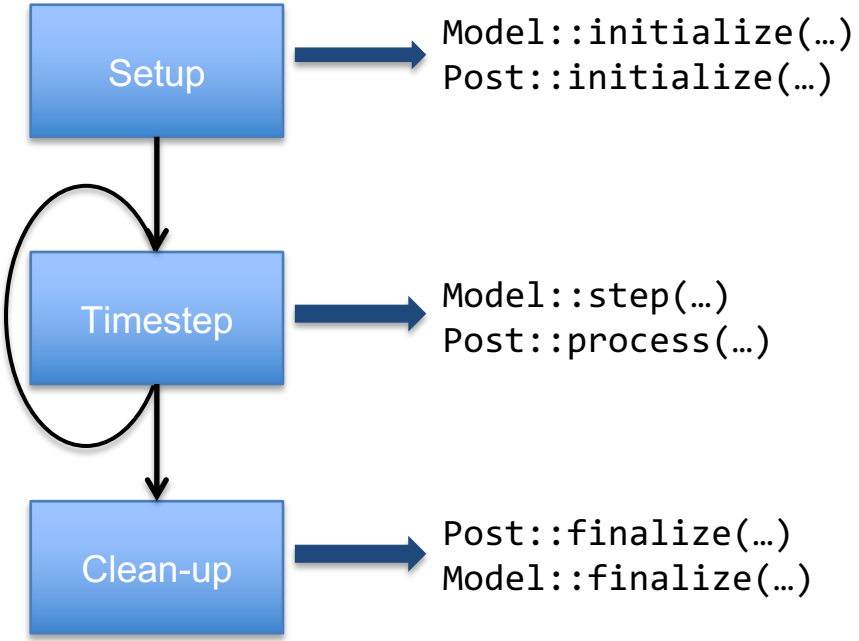
- Naive approach:
  - One object for each term of the cost function.
  - Compute each term (or gradient) and add them together.
  - Problem: The model is run several times (Jo, Jc)
- Another naive approach:
  - Run the model once and store the full 4D state.
  - Compute each term (or gradient) and add them together.
  - Problem: The full 4D state is too big (for operational use).

Cost Function Design

$$J(x) = J_o + J_b + J_c$$

- A feasible approach:
  - Run the model once.
  - Compute each term (gradient) as the model is running (in the postprocessor).
  - Add all the terms together.

Model forecast



Between model “steps” OOPS calls postprocessors

- OOPS manages when post-processors are called
- Post-processing removed from model code (**separation of concerns**)

Post-processors isolate data assimilation from the model (**separation of concerns**)

- Computing simulated observations H(x)
- Jc-DFI, …

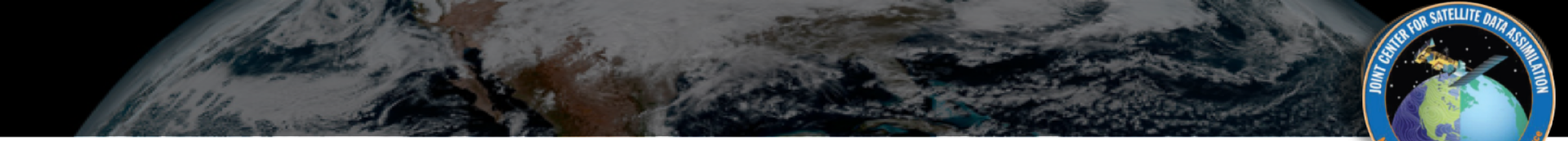
Post-processors do not modify the State

Post Processors

- PostProcessors are called regularly during model integration
  - The presence or not of post processing does not affect the forecast
  - PostProcessors isolate the model code from many other unrelated aspects (separation of concerns)
- Examples:
  - Output of forecast fields
  - Generation of products for users
  - Computation of filtered state (for DFI)
  - Generation of trajectory for linearized models
  - Generation of simulated observations
  - Diagnostics

Cost Function Design

- One class for each term (more flexible).
- Call a method on each object on the fly while the model is running.
  - Uses the PostProcessor structure.
  - Finalize each term and add the terms together at the end.
  - Saving the model linearization trajectory is also handled by a PostProcessor.
- Each formulation derives from an abstract CostFunction base class.
  - Code duplication: it was decided to keep 3D-Var and 4D-Var for readability.
- The terms can be re-used (or not), 4D-En-Var was added in a few hours.
  - OO is not magic and will not solve scientific questions by itself.
  - Scientific questions (localization) remain but scientific work can start.
  - Weeks of work would have been necessary in traditional systems.



To be continued tomorrow: From Abstract to Concrete