Joint Effort for Data assimilation Integration

Object Oriented Prediction System (OOPS)

Yannick Trémolet

JCSDA

04 June 2018

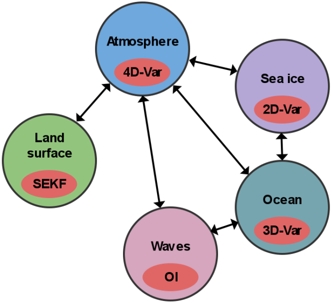
Outline

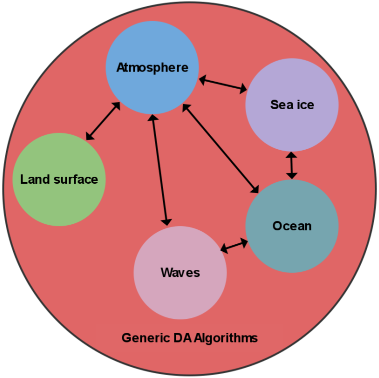
Evolution of Forecasting

- The expectations of society for better weather (and related) forecasts are pushing us to account for more of the Earth system.
- Science and models have progressed in many areas:
  - Atmosphere,
  - Land surface,
  - Ocean,
  - Sea ice,
  - Atmospheric composition...
- Each model is becoming more and more complex as science progresses.
- The models are becoming more and more coupled to account for interactions between all these aspects.

Earth System Data Assimilation

- Data assimilation systems have been developped for each model.





- Coupled data assimilation requires some common infrastructure.

Evolution of Data Assimilation

- Data assimilation algorithms have become very complex over the years:
  - Number and types of observations,
  - Minimisation and preconditioning,
  - Observation bias correction,
  - Sophisticated TL/AD models,
  - Sophisticated observation operators,
  - Wavelet J_b...
  - It is still being developped and improved (weak constraint).
- Today's best data assimilation algorithms are hybrid.
  - Ensemble DA (EDA, 4D-En-Var, EVIL, EnKF) system for computing background error covariances and initializing ensemble forecasts,
  - Variational DA system to provide the high resolution (or *best*) analysis.
- Data assimilation systems have become so complex that comparing all options is almost impossible.

Scalability

Scalability is the ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth (wikipedia)

- This applies to running on increasly large number of processors
- It also applies to:
  - the number of code units
  - the number of developers/users/institutions involved

Outline

Flexible

- It should be easy to modify the system (new science, new functionality, better scalability...)

- A requirement is that a change to one aspect should not imply changes all over the place.
  - No code duplication: same modification in many places but also difficult to find and leads to bugs.
  - No global variables: a modification might have unforeseen consequences anywhere.
  - Think of it in terms of *locality* in the source code (as opposed to discontinous code that jumps all over the place).

Reliable

- The code must run without crashing.
- Additional aspects of reliablity are application dependent. For a complex system, the code must do what the user thinks it does:
  - Many experiments are wasted because it is not always the case.
  - The code must run with the user supplied value (namelist, json, yaml...) or abort.
- A controlled abort with a clear error message is not a crash: it saves computer and user time (our time).
- Lots of testing:
  - Internal consistency and correctness of results (this is not meteorological evaluation),
  - Mecanism to run all the tests easily,
  - Tests run automatically on push to source repository.

Modular

- The weather forecasting problem can be broken into manageable pieces:
  - Data assimilation (or ensemble prediction) can be described without knowing the specifics of a model or observations.
  - Minimisation algorithms can be written without knowing the details of the matrices and vectors involved.
  - Development of a dynamical core on a new model grid should not require knowledge of the data assimilation algorithm.
- Separation of concerns:
  - All aspects exist but scientists focus on one aspect at a time.
  - Different concepts should be treated in different parts of the code.
- Unfortunately, in most cases, Fortran modules don't lead to modular codes.

Object-Oriented Programming

- We need a very flexible, reliable, efficient, readable and modular code.
  - Readability improves staff efficiency: it is as important as computational efficiency (it's just more difficult to measure).
  - Modularity improves staff scalability: it is as important as computational scalability (it's just more difficult to measure).

- This is not specific to the IFS: the techniques that have emerged in the software industry to answer these needs are called **generic** and **object-oriented** programming.

- Object-oriented programming does not solve scientific problems in itself: it provides a more powerful way to tell the computer what to do.

Outline

1. Scalability and Complexity
2. What can we do?
3. OOPS design
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Applications
4. From old to new

Outline

OOPS Analysis and Design

- What is data assimilation?
  Data assimilation is finding the best estimate (analysis) of the state of the atmosphere (or system of interest) given a previous estimate of the state (background) and recent observations of the system.

OOPS Analysis and Design

- What is data assimilation?
  Data assimilation is finding the best estimate (analysis) of the state of the atmosphere (or system of interest) given a previous estimate of the state (background) and recent observations of the system.

OOPS Analysis and Design

- What is data assimilation?
  Data assimilation is finding the best estimate (analysis) of the state of the atmosphere (or system of interest) given a previous estimate of the state (background) and recent observations of the system.
- States :
- Observations :

OOPS Analysis and Design

- What is data assimilation?
  Data assimilation is finding the best estimate (analysis) of the state of the atmosphere (or system of interest) given a previous estimate of the state (background) and recent observations of the system.
- States properties:
  - Input, output (raw or post-processed).
  - Access values.
  - Move forward in time (using the model).
  - Copy, assign.
- Observations :

OOPS Analysis and Design

- What is data assimilation?
  Data assimilation is finding the best estimate (analysis) of the state of the atmosphere (or system of interest) given a previous estimate of the state (background) and recent observations of the system.
- States properties:
  - Input, output (raw or post-processed).
  - Access values.
  - Move forward in time (using the model).
  - Copy, assign.
- Observations properties:
  - Input, output.
  - Simulate observation from a state (observation operator).
  - Copy, assign.

OOPS Analysis and Design

- What is data assimilation?
  Data assimilation is finding the best estimate (analysis) of the state of the atmosphere (or system of interest) given a previous estimate of the state (background) and recent observations of the system.

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

- Observations properties:
  - Input, output.
  - Simulate observation from a state (observation operator).
  - Copy, assign.

- We don't need to know how these operations are performed, how the states are represented or how the observations are stored.

OOPS Analysis and Design

$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{x}_0 - \mathbf{x}_b)^T \mathbf{B}^{-1}(\mathbf{x}_0 - \mathbf{x}_b) + \frac{1}{2}\sum_{i=0}^{n}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]^T \mathbf{R}_i^{-1}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]$$

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

OOPS Analysis and Design

$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{x}_0 - \mathbf{x}_b)^T \mathbf{B}^{-1}(\mathbf{x}_0 - \mathbf{x}_b) + \frac{1}{2}\sum_{i=0}^{n}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]^T \mathbf{R}_i^{-1}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]$$

- Increments:
  - Basic linear algebra operators,
  - Evolve forward in time linearly and backwards with adjoint.
  - Compute as difference between states, add to state.
- Departures:
  - Basic linear algebra operators,
  - Compute as difference between observations, add to observations,
  - Compute as linear variation in observation equivalent as a result of a variation of the state (linearized observation operator).
  - Output (for diagnostics).

OOPS Analysis and Design

$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{x}_0 - \mathbf{x}_b)^T \mathbf{B}^{-1}(\mathbf{x}_0 - \mathbf{x}_b) + \frac{1}{2}\sum_{i=0}^{n}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]^T \mathbf{R}_i^{-1}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]$$

- Increments:
  - Basic linear algebra operators,
  - Evolve forward in time linearly and backwards with adjoint.
  - Compute as difference between states, add to state.
- Departures:
  - Basic linear algebra operators,
  - Compute as difference between observations, add to observations,
  - Compute as linear variation in observation equivalent as a result of a variation of the state (linearized observation operator).
  - Output (for diagnostics).
- Covariance matrices:
  - Setup,
  - Multiply by matrix (and possibly its inverse).

OOPS Abstract Design

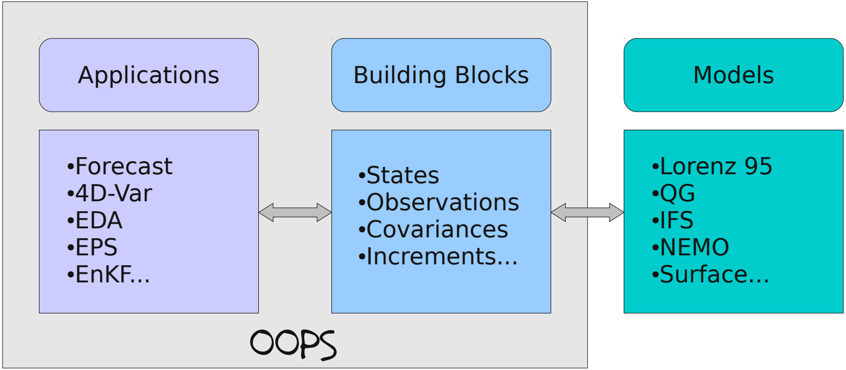
$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{x}_0 - \mathbf{x}_b)^T \mathbf{B}^{-1}(\mathbf{x}_0 - \mathbf{x}_b) + \frac{1}{2}\sum_{i=0}^{n}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]^T \mathbf{R}_i^{-1}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]$$

- The 4D-Var problem, and the algorithm to solve it, can be described with a very limited number of entities:
  - Vectors: $\mathbf{x}$, $\mathbf{y}$, $\mathbf{g}$ and $\delta\mathbf{x}$.
  - Covariances matrices: $\mathbf{B}$, $\mathbf{R}$ (and eventually $\mathbf{Q}$).
  - Two operators and their linearised counterparts: $\mathcal{M}$, $\mathbf{M}$, $\mathbf{M}^T$, $\mathcal{H}$, $\mathbf{H}$, $\mathbf{H}^T$.
- All data assimilation schemes manipulate the same limited number of entities.
- For future (unknown) developments these entities should be easily available and reusable.
- We have not mentioned any details about how any of the operations are performed, how data is stored or what the model represents.

OOPS Abstract Design

- OOPS is independent of the model and the physical system it represents.
- Flexibility (including yet unknown future development) requires that this goes both ways.
- The Models do not know about the high level algorithm currently being run:
  - All actions are driven by the top level code,
  - All data, input and output, is passed by arguments.
- Models interfaces must be general enough to cater for all cases, and detailed enough to be able to perform the required actions.
- OOPS currently stops at the level of the calls to the forecast model and observation operators but the same principle could be applied at any level.

OOPS Abstract Design



- The high levels Applications use abstract building blocks.
- The Models implement the building blocks.
- OOPS is independent of the Model being driven.

Outline

State-Observations Interactions

- Two classes make the link between the model and observation spaces:
  - Locations
  - ModelAtLocations
- The computation of observations equivalents is done in a `PostProcessor`:
  1. Ask the Observations for a list of locations where there are observations (at the current time)
  2. Ask the State for the model values at these locations
  3. Ask the ObsOperator to compute the observations equivalents given the model values at observations locations.

State-Observations Interactions

- Two classes make the link between the model and observation spaces:
  - Locations
  - ModelAtLocations
- The computation of observations equivalents is done in a `PostProcessor`:
  1. Ask the Observations for a list of locations where there are observations (at the current time)
  2. Ask the State for the model values at these locations
  3. Ask the ObsOperator to compute the observations equivalents given the model values at observations locations.
- Last step can be performed on the fly or in the finalize method (memory vs. load balancing).
- The traits ensure the arguments types are compatible. There is no magic interpolation from any grid to any location in OOPS.
- Preserves encapsulation (model grid not visible in observation operator).

State-Observations Interactions

- Two classes make the link between the model and observation spaces:
  - Locations
  - ModelAtLocations
- The computation of observations equivalents is done in a `PostProcessor`:
  1. Ask the Observations for a list of locations where there are observations (at the current time)
  2. Ask the State for the model values at these locations
  3. Ask the ObsOperator to compute the observations equivalents given the model values at observations locations.
- Last step can be performed on the fly or in the finalize method (memory vs. load balancing).
- The traits ensure the arguments types are compatible. There is no magic interpolation from any grid to any location in OOPS.
- Preserves encapsulation (model grid not visible in observation operator).
- But it's up to each model implementation: OOPS does not prevent copying the full State in the GOM...

Cost Function Design

- 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 (J_o, J_c, J_q)

Cost Function Design

- 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 (J_o, J_c, J_q)
- 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 us).

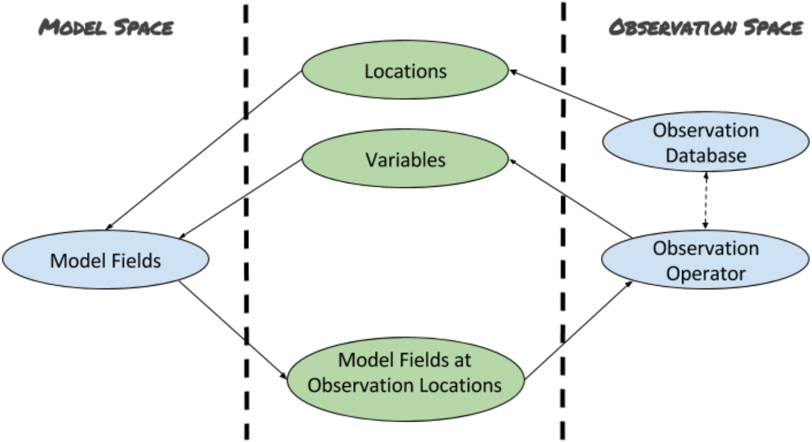
Cost Function Design

- 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 (J_o, J_c, J_q)
- 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 us).
- A feasible approach:
  - Run the model once.
  - Compute each term (or gradient) on the fly while the model is running.
  - Add all the terms together.

Cost Function Implementation

- 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 already in place (observer pattern).
  - Finalize each term and add the terms together at the end.
  - Saving the model linearization trajectory is also the responsibility of a PostProcessor.
- Each formulation derives from an abstract CostFunction base class.
  - Code duplication between strong and weak constraint 4D-Var: use in the same derived class (weak constraint) or write the weak constraint 4D-Var as a sum of strong constraint terms for each sub-window.
  - It was decided to keep 3D-Var and 4D-Var for readability reasons.
- The terms can be re-used (or not), 4D-Ens-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 the IFS.

Generic UFO



- Classes have to be compatible
- Generic but not polymorphic

Outline

From (IFS, GSI, NavDAS...) to (OOPS, JEDI)

- The main idea is to keep the computational parts of the existing code and reuse them in a re-designed flexible structure.
- This can be achieved by a top-down and bottom-up approach.
  - From the top: Develop a new, modern, flexible structure (C++).
  - From the bottom: Progressively create self-contained units of code (Fortran).
  - Put the two together: Extract self-contained parts of the IFS and plug them into OOPS.
- From a Fortran point of view, this implies:
  - No global variables,
  - Control via interfaces (derived types passed by arguments).
- This is done at high level in the code.
  - It complements work on code optimisation done at lower level.
- The OO layer developed for the simple models is not only a proof of concept: the same code is re-used to drive the IFS (generic).