Software Infrastructure for Numerical Weather Prediction

OOPS Project

Yannick Trémolet

ECMWF

November 2015

Outline

1 Complexity
   - Computing complexity
   - Model complexity
   - Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4 From IFS to OOPS
Outline

1 Complexity
   - Computing complexity
     - Model complexity
     - Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4 From IFS to OOPS
The Scalability Question

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).

Reducing power consumption will require complex architectures.
• Implement a formal structure at ECMWF to coordinate science and software activities across departments for efficient exa-scale computing/archiving.

• Include and coordinate all components of the system, including data assimilation, model, data pre- and post-processing and archiving.
Outline

1 Complexity
   - Computing complexity
   - Model complexity
   - Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4 From IFS to OOPS
Evolution of Forecasting: Earth System Modelling

- 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.
An example: IFS complexity

It means growth of maintenance, development costs, and number of bugs.
Data assimilation systems have been developed for each model.

Coupled data assimilation requires a common framework.
Outline

1 Complexity
   - Computing complexity
   - Model complexity
   - Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4 From IFS to OOPS
Evolution of Data Assimilation

- 4D-Var has been the main staple of data assimilation at ECMWF since 1997.

- The algorithm has progressed to become more complex over the years. It is still being developed and improved (weak constraint, saddle point).

- ECMWF uses an ensemble of 4D-Vars to estimate background error statistics. There are alternatives:
  - EnKF, 4D-En-Var, EVIL...

- Today’s best data assimilation algorithms are hybrid:
  - Again, there are a number of options for combining the variational and ensemble aspects

- The data assimilation scene is uncertain but complex...
Complexity

- Computing

- Models

- Data Assimilation

More and more people are working with the same codes

Complexity needs to be managed
Outline

1 Complexity
   - Computing complexity
   - Model complexity
   - Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4 From IFS to OOPS
Flexible + Reliable = Modular?

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

- **Reliability:**
  - The code must run without crashing.
  - For a system like the IFS, the code must do what the user thinks it does.

- **Modularity is the answer!**

- And 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.
  - Development of a dynamical core on a new model grid should not require knowledge of the data assimilation algorithm.
Flexible + Reliable = Modular?

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

- **Reliability:**
  - The code must run without crashing.
  - For a system like the IFS, the code must do what the user thinks it does.

- **Modularity is the answer!**

- And 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.
  - Development of a dynamical core on a new model grid should not require knowledge of the data assimilation algorithm.

- 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.

- It promotes 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.
Outline

1 Complexity
   - Computing complexity
   - Model complexity
   - Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4 From IFS to OOPS
Outline

1. Complexity
   - Computing complexity
   - Model complexity
   - Data assimilation complexity

2. What can we do?

3. Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4. From IFS to OOPS
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: \( \mathbf{M}, \mathbf{M}^T, \mathbf{H}, \mathbf{H}^T \).

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.
The high levels Applications use abstract building blocks.

The Models implement the building blocks.

OOPS is independent of the Model being driven.

Models interfaces must be general enough to cater for all cases, and detailed enough to be able to perform the required actions.
Outline

1 Complexity
   - Computing complexity
   - Model complexity
   - Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4 From IFS to OOPS
OOPS Classes

- OOPS requires a consistent set of classes that work together with predefined interfaces:
  - In model space:
    1. Geometry
    2. State
    3. Increment
    4. Model
    5. LinearModel (TLM and adjoint)
  - In observation space:
    6. ObservationSpace
    7. ObsVector
    8. ObsOperator
    9. LinearObsOperator
  - To make the link:
    10. Locations
    11. ModelAtLocations
    12. Variables
  - Covariance matrices (if generic ones are not used):
    13. Model space (B and Q)
    14. Observation space (R)
    15. Localization (4D-Ens-Var)

- Approximately 100 methods to be implemented (in Fortran or not).
- Observation and model errors (biases) can also be defined.
Model Trait Definition

The trait is used as a template argument <MODEL>: compile time polymorphism.
Model Trait Definition

The trait is used as a template argument `<MODEL>`: compile time polymorphism.

```cpp
struct IfsTraits {
    typedef ifs::GeometryIFS Geometry;
    typedef ifs::StateIFS State;
    typedef ifs::IncrementIFS Increment;
    typedef ifs::ModelIFS Model;
    typedef ifs::LinearModelIFS LinearModel;
    typedef ifs::ODBwrapper ObsSpace;
    typedef ifs::AllObs ObsOperator;
    typedef ifs::AllObsTLAD LinearObsOperator;
    typedef ifs::ObsVector ObsVector;
    typedef ifs::ObsBias ObsAuxControl;
    typedef ifs::ObsBiasIncrement ObsAuxIncrement;
    typedef ifs::ObsBiasCovariance ObsAuxCovariance;
    typedef ifs::GomsIFS ModelAtLocations;
    typedef ifs::LocationsIFS Locations;
    typedef ifs::VariablesIFS Variables;
    typedef ifs::ErrorCovariance3D Covariance;
    typedef ifs::LocalizationMatrixIFS LocalizationMatrix;
};
```
Running a forecast

```
template<typename MODEL>
void Model<MODEL>::forecast(const ModelAuxCtrl_ & mctl,
    const util::Duration & len, PostProcessor<State_> & post) {
    const util::DateTime end(validTime() + len);
    post.initialize(validTime(), end, model_.timestep());
    this->init(model_);
    post.process(*this);

    while (validTime() < end) {
        this->step(model_, mctl);
        post.process(*this);
    }
    post.finalize();
}
```

- forecast calls the PostProcessors at each time step (Observer pattern).
- PostProcessors are very generic: I/O, FullPos, print information...
- It is the responsibility of the PostProcessors to know when and what actions are needed, not of the model.
- The responsibility of the model is to move the state in time, nothing else.
Outline

1 Complexity
   - Computing complexity
   - Model complexity
   - Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4 From IFS to OOPS
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, using the PostProcessor structure already in place.
  - Finalize each term and add the terms together at the end.
Outline

1 Complexity
   • Computing complexity
   • Model complexity
   • Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   • OOPS Design: Abstract Level
   • Implementing the Abstract Design: Building Blocks
   • Implementing the Abstract Design: Applications
   • PyOOPS

4 From IFS to OOPS
Like the Fortran code, the suite definitions and scripts have become more and more difficult to maintain and develop.

Complexity will keep increasing in the future:
- Long overlapping 4D-Var windows,
- Hybrid data assimilation (EDA and DA coupled two-ways),
- Coupled ocean-atmosphere models...

The suite definitions and scripts define the application at the highest level.
- We should think of them as part of the “system”.

Three levels are mixed together in the suite definitions and scripts:
- The model (IFS, NEMO...), although the top level of OOPS is generic,
- The “scientific” description of the cycling,
- The workflow “technical” specificity (SMS, ecflow, ...).

The three levels could be, and should be, isolated from each other.
A prototype has been implemented in python to test the approach.

It is organised around tasks whose input and outputs are metadata objects.

The metadata objects are also used by the workflow to generate the triggers.

class Analysis(CompositeTask):
    def compose(self):
        window = self.input('window')

        bg = self.bgfc(window=window)
        obs = self.fetchobs(window=window)
        (an, fb) = self.an4dvar(bg=bg, obs=obs, window=window)

        self.archive_bg(data=bg)
        self.archive_fb(data=fb)

        self.set_output('an', an)

Note that GetBackground is a composite task as well!

The workflow (ecFlow) is abstracted from the suite definition.
Abstracting the workflow

- Scientists should think as if writing any algorithm.

- Executing the (python) code generates the suite (and scripts).
  - Each component can generate a single task or a family.
  - The workflow is chosen when running the python program.
  - A simple workflow can run the tasks on the fly (toy system on a laptop).

- The workflow can be specialized for Operations to control when the observations are retrieved and the analysis cycle started.

- Everything else is the same: More can be shared between research and operations.

  The key is again separation of concerns
Outline

1 Complexity
   - Computing complexity
   - Model complexity
   - Data assimilation complexity

2 What can we do?

3 Object Oriented Prediction System
   - OOPS Design: Abstract Level
   - Implementing the Abstract Design: Building Blocks
   - Implementing the Abstract Design: Applications
   - PyOOPS

4 From IFS to OOPS
From IFS to OOPS

- The main idea is to keep the efficient 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.
OOPS Summary

- Code components are independent:
  - Components can easily be developed in parallel.
  - Their complexity decreases: less bugs and easier testing and debugging.

- Improved flexibility:
  - Develop new data assimilation (and other) science.
  - Prepares DA for potentially dramatic changes the model (scalability).
  - Explore and improve data assimilation scalability.
  - Changes in one application do not affect other applications.
  - Ability to handle different models opens the door for coupled DA.

- Other projects in the scalability programme are following a similar approach, using OOP (and C++) to abstract low level data structures and architectures.

- The main difficulties are not technical but human...