ECMWF

OOPS: Object-Oriented Prediction System
Restructuring the IFS

Yannick Trémolet

ECMWF

November 18, 2009

Outline

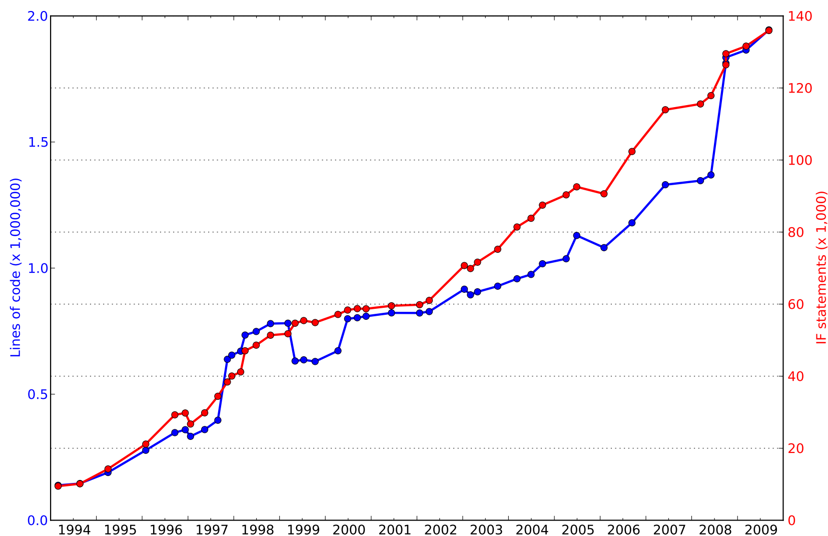
1. Current state of the IFS
2. Modernising the IFS
3. Preliminary study

Motivation

ECMWF

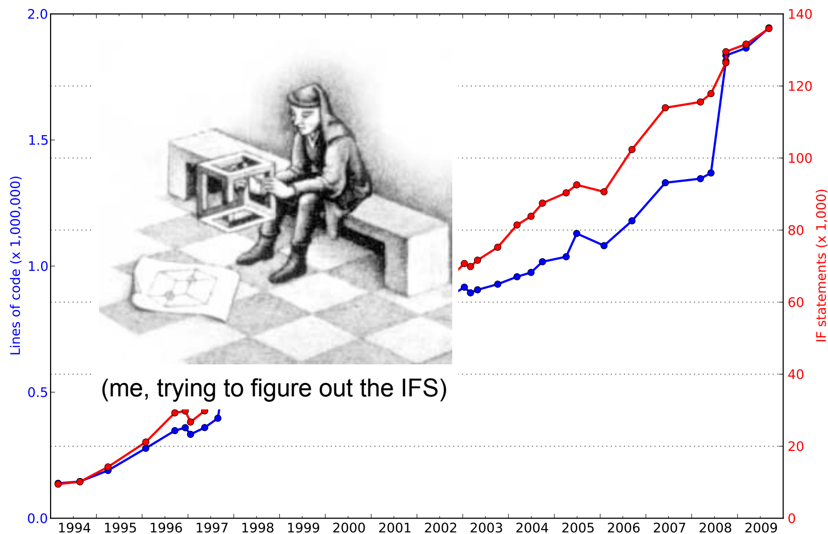
- The IFS code is more than twenty years old and, over this period, has reached a very high level of complexity.
- It is difficult to modify one configuration without breaking others.
- The maintenance cost has become very high.
- New cycles take longer and longer to create and debug.
- There is a long, steep learning curve for new scientists and visitors.
- Collaboration with external groups can be difficult.
- It is becoming a barrier to new scientific developments such as long window weak constraints 4D-Var.

IFS code growth



Unfortunately, it's not an investment: It's growth of costs, not of benefits.

IFS code growth



Unfortunately, it's not an investment: It's growth of costs, not of benefits.

Example: Data Assimilation

ECMWF

- The data assimilation method in the IFS is 4D-Var which comprises the minimisation of the cost function:

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

- 4D-Var is a complex and computationally very demanding problem.
- However, the 4D-Var problem, and the algorithm to solve it, can be described with a very limited number of entities:
  - Vectors: $\mathbf{x}$, $\mathbf{y}$, $\mathbf{g}$, $\delta\mathbf{x}$ and χ.
  - Covariances matrices: **B**, **R** (and eventually **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 (KF, EnKF, PSAS...).
- For future scientific developments these entities should be easily available and reusable.

Current situation in the IFS

ECMWF

- Most high level routines don't have arguments (global variables).
  - Algorithms not envisaged at the outset (25+ years ago) are extremely difficult to implement.
- Setup routines are separated from the rest of the code.
  - All variables have to be accessible from four places (*module*, namelist, setup, subroutine) instead of one.
- Entities are not always independent.
  - $\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}$ is one piece (jumble) of code.
- No structure exists to manipulate vectors in observation space (or in model space!).
  - Observation space algorithms are virtually impossible to implement.
- The nonlinear model $\mathcal{M}$ can only be integrated once per execution.
  - Algorithms that require several calls to $\mathcal{M}$ can only be written at script level.

Other areas can benefit from flexibility

- Singular vector computation.
- Observation operators (SAT, verification).
- J_b for limited area model.
- Ensemble data assimilation.
- Grid-point model, or change of model grid (A-grid to C-grid).
- Code validation, for example generic adjoint tests.
- Restart 4D-Var.

Outline

1. Current state of the IFS
2. Modernising the IFS
3. Preliminary study

An approach to modernize the IFS

ECMWF

- The entities described earlier should be easily recognisable and reusable in the code.
- Data assimilation can be described without knowing the details of the components being used: the algorithm should be separated from the details of the implementation of the objects it deals with.
- Modularity is the answer!

An approach to modernize the IFS

ECMWF

- The entities described earlier should be easily recognisable and reusable in the code.
- Data assimilation can be described without knowing the details of the components being used: the algorithm should be separated from the details of the implementation of the objects it deals with.
- Modularity is the answer! Yes, but...
- It means more than using the keyword `module`: in the IFS, it is mostly an equivalent to common blocks used to define global variables. Using `modules` leads to the opposite of a modular code!
  - The conversion from `commons` to `modules` was performed mostly automatically by scripts. There were no changes in the design.
- Information hiding and abstraction are important.

What is modularity?

ECMWF

- Decomposability: break the problem into small enough independent less complex subproblems.
- Composability: elements can be freely combined to produce a new system.
- Understandability: elements can be understood without knowing the others.
- Continuity: a small change in the problem specification triggers a change in just one (or few) module(s).
- Protection: an error does not propagate to other modules.
- Some rules for modularity:
  - Few interfaces,
  - Small interfaces,
  - Explicit interfaces,
  - Information hiding.

A comment about modularity

- From IFS coding norms:
  CTRL(16) Considering a namelist, its content should be saved in a specic data module and initialized in a specic subroutine. All three should be named with the same radical. For example: the content of the namelist NAMCT0 is saved in the module YOMCT0 and initialized in a subroutine SUCT0.

A comment about modularity

ECMWF

- From IFS coding norms:
  CTRL(16) Considering a namelist, its content should be saved in a specic data module and initialized in a specic subroutine. All three should be named with the same radical. For example: the content of the namelist NAMCT0 is saved in the module YOMCT0 and initialized in a subroutine SUCT0.
- B. Meyer, Object-Oriented software construction, 2007, p. 41:
  - *A typical counter example* [to modularity] *is any method encouraging you to include, in each software system that you produce, a global initialization module* [Not in the Fortran sense].

A comment about modularity

ECMWF

- From IFS coding norms:
  CTRL(16) Considering a namelist, its content should be saved in a specic data module and initialized in a specic subroutine. All three should be named with the same radical. For example: the content of the namelist NAMCT0 is saved in the module YOMCT0 and initialized in a subroutine SUCT0.

- B. Meyer, Object-Oriented software construction, 2007, p. 41:
  - *A typical counter example* [to modularity] *is any method encouraging you to include, in each software system that you produce, a global initialization module* [Not in the Fortran sense].

- Why is the norm so bad?
  - This introduces unecessary dependencies.
  - Each modification has to be repeated in all this places.

A comment about modularity

- From IFS coding norms:
  CTRL(16) Considering a namelist, its content should be saved in a specic data module and initialized in a specic subroutine. All three should be named with the same radical. For example: the content of the namelist NAMCT0 is saved in the module YOMCT0 and initialized in a subroutine SUCT0.

- B. Meyer, Object-Oriented software construction, 2007, p. 41:
  - *A typical counter example* [to modularity] *is any method encouraging you to include, in each software system that you produce, a global initialization module* [Not in the Fortran sense].

- Why is the norm so bad?
  - This introduces unecessary dependencies.
  - Each modification has to be repeated in all this places.

- The currrent IFS coding norms are too restrictive. In this case, they forbid good programming practices.

- Each module should take care of its own initialisation.

Modularity and Object Oriented Programming

ECMWF

- The general technique that has emerged in the software industry to address the needs for flexibility, reusability, reliability and efficiency is object-oriented programming.
- One key idea of Object-Oriented programming is to organise the code around the data, not around the algorithms.

Modularity and Object Oriented Programming

ECMWF

- The general technique that has emerged in the software industry to address the needs for flexibility, reusability, reliability and efficiency is object-oriented programming.
- One key idea of Object-Oriented programming is to organise the code around the data, not around the algorithms.
- Does it make sense to rethink the design of the IFS in that framework?

Programming Languages

ECMWF

- Fortran 2003:
  - First compilers are becoming available,
  - First impression is that the language is cumbersome.
  - Easy-ish transition from existing code (at the risk of adopting existing solutions only because it is easy).
- C++:
  - Widely used language for supercomputing (outside meteorology),
  - Transition is sligthly more complex,
  - Might require more training for existing staff.
- Python:
  - Easiest language to use for prototyping,
  - Performance could be an issue,
  - Could be used for high level structure (cycling, outer loops).
- Chapel, X10, Fortress...
  - Some good aspects but not portable.

Outline

1 Current state of the IFS

2 Modernising the IFS

3 Preliminary study

Scope of the preliminary study

ECMWF

- Preliminary feasibility study with toy models: validation of ideas and evaluation of programming languages for our purpose.
  - Develop the entire incremental 4D-Var algorithm for very simple models.
  - Include built-it validation methods (technical and scientific).
  - The result should be a 4D-Var framework into which we could plug a variety of models (Lorenz, Lorenz 95 and QG so far).
  - Simple system has been developped in Fortran 2003, C++ and C++/F90.
- Answer the questions:
  - Do we need an object-oriented design?
  - What would be a good OO design for our system?
  - Is Fortran 2003 the best choice for the future?
  - Can it reasonably be applied to the IFS?

Object Oriented Programming in the IFS

- One key idea of Object-Oriented programming is to organise the code around the data, not around the algorithms.

Object Oriented Programming in the IFS

ECMWF

- One key idea of Object-Oriented programming is to organise the code around the data, not around the algorithms.

- Data type in traditional code (*maximum information at the top*):
  - Include all possible components at the highest level in type definition,
  - Each component is switched on/off whenever the type is used,
  - Adding a component means that the code has to be modified in most places where the type is accessed.

Object Oriented Programming in the IFS

ECMWF

- One key idea of Object-Oriented programming is to organise the code around the data, not around the algorithms.
- Data type in traditional code (*maximum information at the top*):
  - Include all possible components at the highest level in type definition,
  - Each component is switched on/off whenever the type is used,
  - Adding a component means that the code has to be modified in most places where the type is accessed.
- Data type (class) in object-oriented code (*minimum information at the top*):
  - High level class contains no details.
  - Details are added at a lower levels when they become necessary for the implementation.
  - Adding a component requires modifying the code only where the component is used. There is no need to modify code that cannot see a component!

OOPS: Basic classes

- 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: Basic classes

ECMWF

- 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: Basic classes

- 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: Basic classes

ECMWF

- 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).
  - Assign,
  - Move forward in time (propagate using the model).

- Observations :

OOPS: Basic classes

ECMWF

- 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).
  - Assign,
  - Move forward in time (propagate using the model).

- Observations properties:
  - Input, output.
  - Assign,
  - Compute observation equivalent from a state (obs_operator).

OOPS: Basic classes

ECMWF

- 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).
  - Assign,
  - Move forward in time (propagate using the model).

- Observations properties:
  - Input, output.
  - Assign,
  - Compute observation equivalent from a state (obs_operator).

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

OOPS: Basic classes

ECMWF

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

- Increments:
  - Basic linear algebra operators,
  - Evolve forward in time linearly and adjoint (propagate_tl, propagate_ad).
  - Add to state.
  - Input, output.

OOPS: Basic classes

ECMWF

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

- Increments:
  - Basic linear algebra operators,
  - Evolve forward in time linearly and adjoint (propagate_tl, propagate_ad).
  - Add to state.
  - Input, output.
- Departures:
  - Compute as difference between observations,
  - Compute as linear variation in observation equivalent as a result of a variation of the state (obs_operator_tl, obs_operator_ad),
  - Output (for diagnostics).

OOPS: Basic classes

ECMWF

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

- Increments:
  - Basic linear algebra operators,
  - Evolve forward in time linearly and adjoint (propagate_tl, propagate_ad).
  - Add to state.
  - Input, output.
- Departures:
  - Compute as difference between observations,
  - Compute as linear variation in observation equivalent as a result of a variation of the state (obs_operator_tl, obs_operator_ad),
  - Output (for diagnostics).
- Covariance matrices:
  - Setup,
  - Multiply by matrix and its inverse (and/or square root).

OOPS: Basic classes

ECMWF

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

- Increments:
  - Basic linear algebra operators,
  - Evolve forward in time linearly and adjoint (propagate_tl, propagate_ad).
  - Add to state.
  - Input, output.
- Departures:
  - Compute as difference between observations,
  - Compute as linear variation in observation equivalent as a result of a variation of the state (obs_operator_tl, obs_operator_ad),
  - Output (for diagnostics).
- Covariance matrices:
  - Setup,
  - Multiply by matrix and its inverse (and/or square root).
  - Often **R** is stored explicitly as a (diagonal) matrix while **B** is implemented as a set of operators but this is not visible.

OOPS: Building-up the system

ECMWF

- The basic classes described previously can be used to build any data assimilation system.
- For example, for an incremental 4D-Var algorithm:
  - Observer (J_o):
    - Iterate over time-steps, compute observation equivalent and move the state forward.
  - Control vectors and change of variable:
    - To speed-up the minimisation, it is preconditionned by the square root of **B**, the unknown variable is a non-physical control vector.
  - Cost function:
    - Takes a control vector in input, does a change of variable, computes the cost function in physical space and performs the adjoint to obtain the gradient (control vector).
  - Minimisation algorithm:
    - Abstract algorithm manipulating abstract vectors and a function to be minimised that takes a vector as input and returns its gradient as a vector.
- This can be developed with any OO language.