

Software Infrastructure for Numerical Weather Prediction

OOPS Project

Yannick Trémolet

ECMWF

JCSDA Seminar - 8 June 2015

OOPS contributors: P. Bauer, W. Deconinck, M. Fisher, A. Geer, M. Hamrud,
O. Marsden, F. Pierfederici, F. Rabier, D. Salmond, J.N. Thépaut, T.
Wilhelmson and others...

Outline

1 Complexity

- Computing complexity
- Model complexity
- Data assimilation complexity

2 What can we do?

3 OOPS design

- OOPS Design: Abstract Level
- Implementing the Abstract Design: Building Blocks
- Implementing the Abstract Design: Applications
- PyOOPS

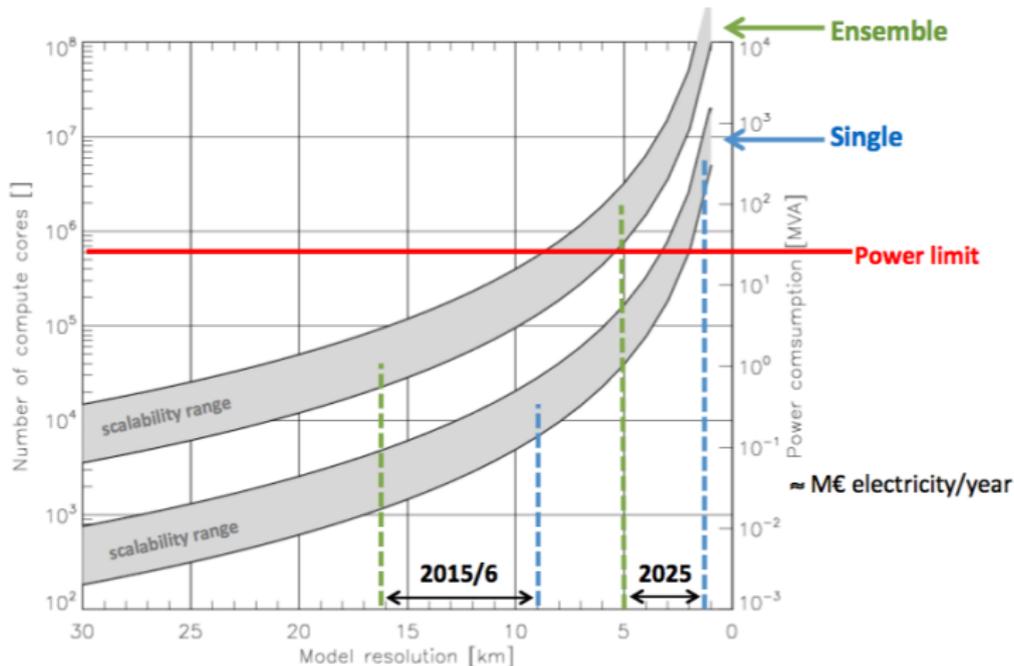
4 From IFS to OOPS

Outline

- 1 Complexity
 - Computing complexity
 - Model complexity
 - Data assimilation complexity
- 2 What can we do?
- 3 OOPS design
 - 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)



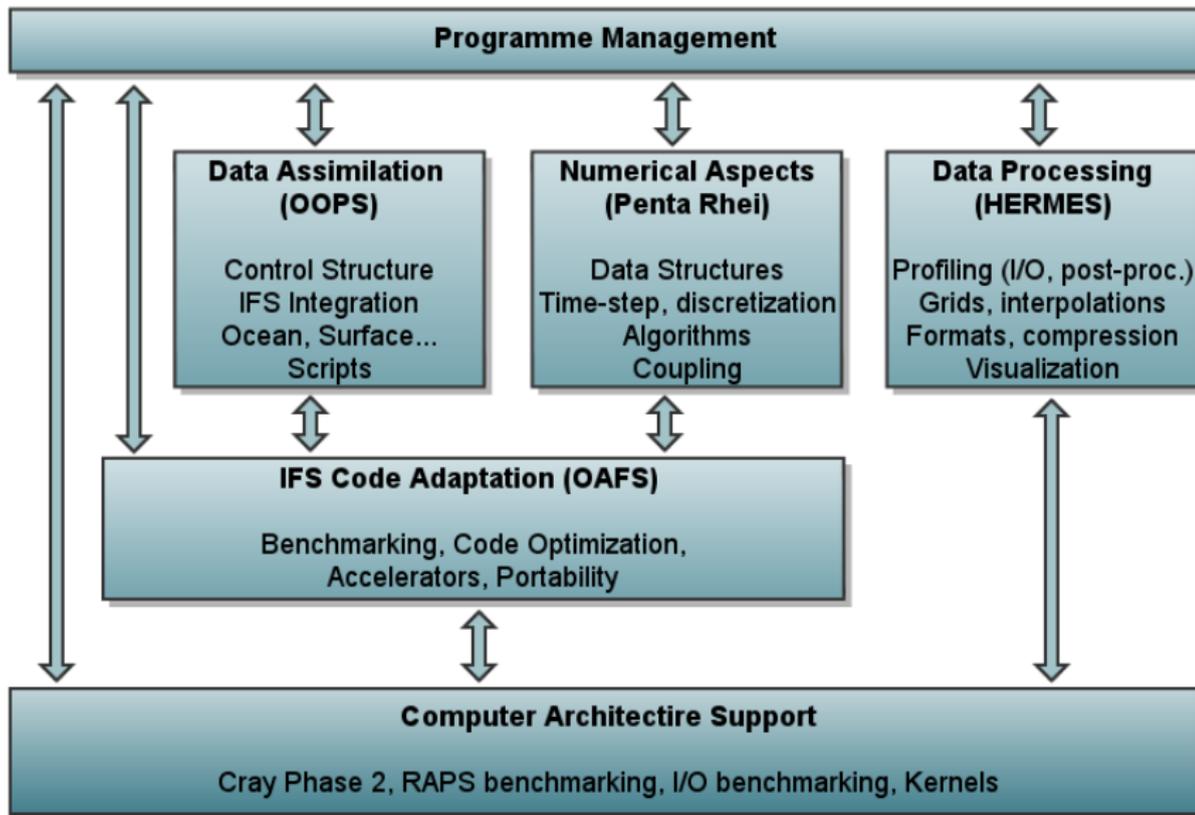
- Future forecast configuration with the current code structure on the next generation HPC will:
 - not complete within operational schedule;
 - cause unaffordable levels of electricity.

- Achieving the highest performance per Watt of energy is critical.

- To cope with the imminent technology challenge a step-change effort not an increment is needed.

- Doing nothing is not an option.

- Implement a formal structure at ECMWF to coordinate science and software activities across departments for efficient exa-scale computing/archiving.
- Coordinate activities with Member States, European HPC facilities, research centres, academia, vendors and international NWP centres.
- Include and coordinate all components of the system, including data assimilation, model, data pre- and post-processing and archiving.
- The main objectives are:
 - to develop the future IFS combining a flexible framework for scientific choices to be made with maximum achievable parallelism,
 - to prepare for expected future hardware technologies and their implications on code structure ensuring efficiency and code readability,
 - to develop environments/metrics for quantitative scalability assessment.
- The success metrics are:
 - efficiency gains in Watts (not FLOPS)
 - efficiency gains in Gbyte/s and Pbyte



Outline

1 Complexity

- Computing complexity
- **Model complexity**
- Data assimilation complexity

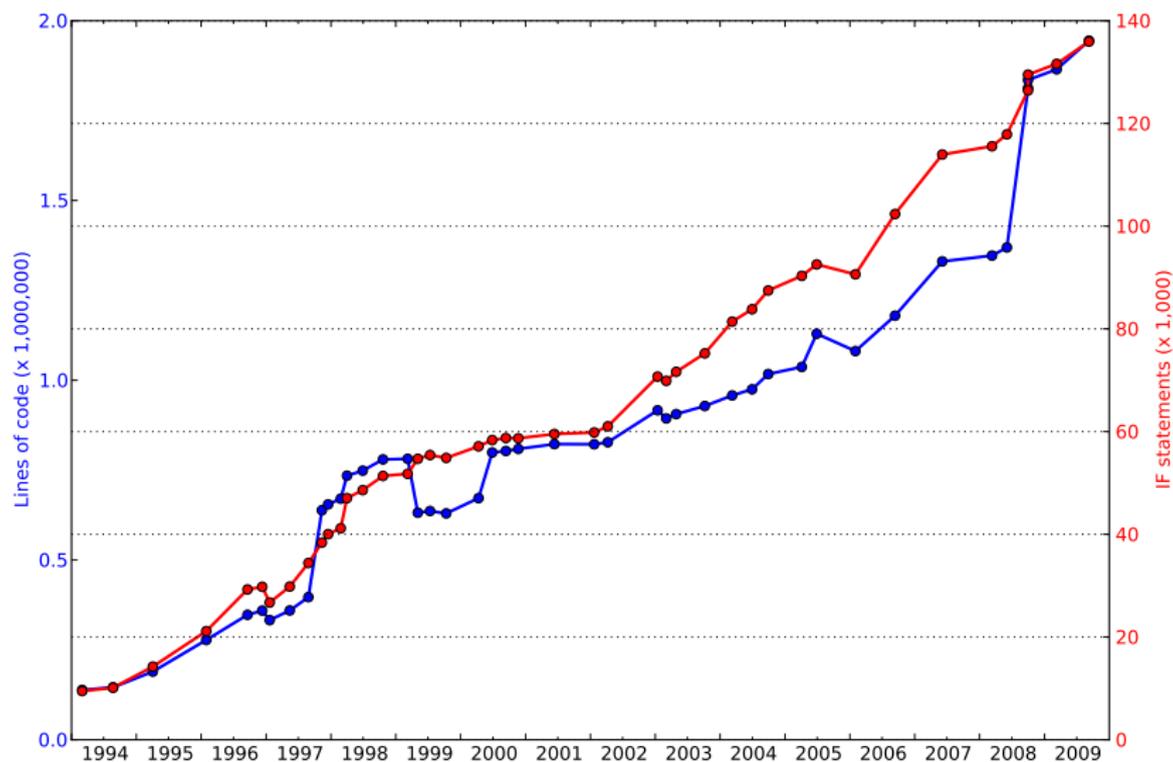
2 What can we do?

3 OOPS design

- OOPS Design: Abstract Level
- Implementing the Abstract Design: Building Blocks
- Implementing the Abstract Design: Applications
- PyOOPS

4 From IFS to OOPS

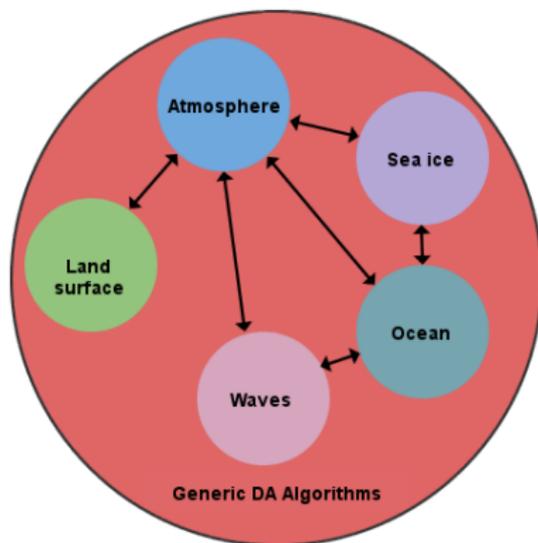
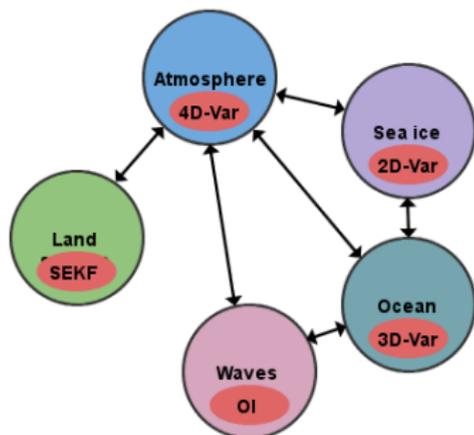
- The expectations of society for better weather (and related) forecasts are pushing us to account for more of the Earth system.
- Science and models have progressed in many areas:
 - Atmosphere,
 - Land surface,
 - Ocean,
 - Sea ice,
 - Atmospheric composition...
- Each model is becoming more and more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



It means growth of maintenance, development costs, and number of bugs.

Earth System Data Assimilation

- Data assimilation systems have been developed for each model.



- Coupled data assimilation requires some common framework.

Outline

1 Complexity

- Computing complexity
- Model complexity
- Data assimilation complexity

2 What can we do?

3 OOPS design

- OOPS Design: Abstract Level
- Implementing the Abstract Design: Building Blocks
- Implementing the Abstract Design: Applications
- PyOOPS

4 From IFS to OOPS

- 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:
 - 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 developed and improved (weak constraint).

- ECMWF uses an ensemble of 4D-Vars to estimate background error statistics.
- Alternative: EnKF
 - Approximation of a Kalman filter with covariances projected on ensemble space, with issues related to localisation and inflation.
 - A research implementation is maintained at ECMWF.
- Alternative: 4D-En-Var
 - Approximation of 4D-Var where time evolution of increments and covariances are projected on ensemble space (with localization),
 - Available in OOPS.
- Alternative: EVIL...
- Many options are open...

- Today's best data assimilation algorithms are hybrid.
 - Ensemble DA system for computing background error covariances,
 - Variational DA system to provide the high resolution (or *best*) analysis.

- There are a number of options for combining them:
 - Use flow dependent \mathbf{B} in 4D-Var,
 - Hybrid gain (combine increments).

- Hybrid data assimilation systems are very complex.
 - Each task is complex,
 - Scheduling of jobs,
 - Data flow,
 - Comparing all options is almost impossible.

Outline

1 Complexity

- Computing complexity
- Model complexity
- Data assimilation complexity

2 What can we do?

3 OOPS design

- OOPS Design: Abstract Level
- Implementing the Abstract Design: Building Blocks
- Implementing the Abstract Design: Applications
- PyOOPS

4 From IFS to OOPS

- 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 discontinuous code that jumps all over the place).

- The code must run without crashing.
- Additional aspects of reliability are application dependent. For a system like the IFS, 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, xml) 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),
 - Mechanism to run all the tests easily,
 - Tests run automatically on push to source repository.

- 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 Complexity

- Computing complexity
- Model complexity
- Data assimilation complexity

2 What can we do?

3 OOPS design

- 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 OOPS design

- **OOPS Design: Abstract Level**
- Implementing the Abstract Design: Building Blocks
- Implementing the Abstract Design: Applications
- PyOOPS

4 From IFS to OOPS

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).
- Interpolate.
- 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).
- Interpolate.
- Move forward in time (using the model).
- Copy, assign.

- Observations properties:

- Input, output.
- Compute observation equivalent from a state (observation operator).
- Copy, assign.

- 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).
 - Interpolate.
 - Move forward in time (using the model).
 - Copy, assign.
- Observations properties:
 - Input, output.
 - Compute observation equivalent 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.

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

- Increments:

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

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

- 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).

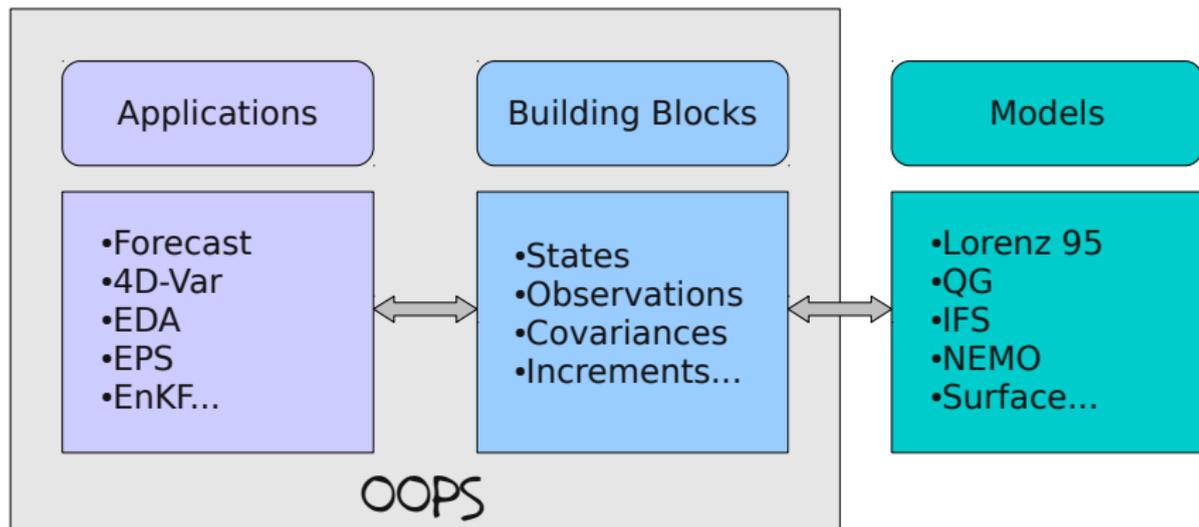
- Covariance matrices:

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

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

- 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 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 OOPS,
 - 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.



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

Outline

1 Complexity

- Computing complexity
- Model complexity
- Data assimilation complexity

2 What can we do?

3 OOPS design

- 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. ModelConfiguration
 5. LinearModel (Trajectory)
 - In observation space:
 6. ObsOperator
 7. ObsAuxControl;
 8. ObsAuxIncrement;
 9. ObsVector
 10. ObsOperatorTrajectory;
 - To make the link:
 11. Locations
 12. ModelAtLocations
 - 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) will be added.

Model Trait Definition

Actual implementation



Name used in OOPS



<code>struct QgTraits {</code>	
<code>typedef qg::QgGeometry</code>	<code>Geometry;</code>
<code>typedef qg::QgState</code>	<code>State;</code>
<code>typedef qg::QgModel</code>	<code>ModelConfiguration;</code>
<code>typedef qg::QgIncrement</code>	<code>Increment;</code>
<code>typedef qg::QgTLM</code>	<code>LinearModel;</code>
<code>typedef oops::NullModelAux</code>	<code>ModelAuxControl;</code>
<code>typedef oops::NullModelAux</code>	<code>ModelAuxIncrement;</code>
<code>typedef qg::QgObservation</code>	<code>ObsOperator;</code>
<code>typedef qg::ObsTrajQG</code>	<code>ObsOperatorLinearizationTrajectory;</code>
<code>typedef oops::NullObsAux</code>	<code>ObsAuxControl;</code>
<code>typedef oops::NullObsAux</code>	<code>ObsAuxIncrement;</code>
<code>typedef qg::ObsVecQG</code>	<code>ObsVector;</code>
<code>typedef qg::LocQG</code>	<code>Locations;</code>
<code>typedef qg::GomQG</code>	<code>ModelAtLocations;</code>
<code>typedef qg::LocalizationMatrixQG</code>	<code>LocalizationMatrix;</code>
<code>};</code>	

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

Model Trait Definition

Actual implementation



Name used in OOPS



<code>struct IfsTraits {</code>	
<code>typedef ifs::GeometryIFS</code>	<code>Geometry;</code>
<code>typedef ifs::StateIFS</code>	<code>State;</code>
<code>typedef ifs::ModelIFS</code>	<code>ModelConfiguration;</code>
<code>typedef ifs::IncrementIFS</code>	<code>Increment;</code>
<code>typedef ifs::LinearModelIFS</code>	<code>LinearModel;</code>
<code>typedef oops::NullModelAux</code>	<code>ModelAuxControl;</code>
<code>typedef oops::NullModelAux</code>	<code>ModelAuxIncrement;</code>
<code>typedef ifs::AllObs</code>	<code>ObsOperator;</code>
<code>typedef ifs::AllObsTraj</code>	<code>ObsOperatorLinearizationTrajectory;</code>
<code>typedef oops::NullObsAux</code>	<code>ObsAuxControl;</code>
<code>typedef oops::NullObsAux</code>	<code>ObsAuxIncrement;</code>
<code>typedef ifs::ObsVector</code>	<code>ObsVector;</code>
<code>typedef ifs::LocationsIFS</code>	<code>Locations;</code>
<code>typedef ifs::GomsIFS</code>	<code>ModelAtLocations;</code>
<code>typedef ifs::LocalizationMatrixIFS</code>	<code>LocalizationMatrix;</code>
<code>};</code>	

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

Increment (L95)

```
class IncrementL95: public FieldL95, public oops::GeneralizedDepartures,
                  private util::ObjectCounter<IncrementL95> {
public:
    static const std::string classname() {return "lorenz95::IncrementL95";}

// Constructor, destructor
    IncrementL95(const Resolution &, const oops::Variables &, const util::DateTim
    IncrementL95(const IncrementL95 &, const Resolution &);
    IncrementL95(const IncrementL95 &, const bool copy = true);
    virtual ~IncrementL95();

// Basic operators
    void diff(const StateL95 &, const StateL95 &);
    IncrementL95 & operator =(const IncrementL95 &);
    IncrementL95 & operator +=(const IncrementL95 &);
    IncrementL95 & operator -=(const IncrementL95 &);
    IncrementL95 & operator *=(const double &);
    void zero();
    void axpy(const double &, const IncrementL95 &, const bool check = true);
    double dot_product_with(const IncrementL95 &) const;
    void schur_product_with(const IncrementL95 &);
    void timeUpdate(const util::Duration &);
```

- The compiler will check the types of the arguments during template instantiation. Run-time polymorphism would require downcasting.

Increment (L95)

```

class IncrementL95: public FieldL95, public oops::GeneralizedDepartures,
                  private util::ObjectCounter<IncrementL95> {
public:
    /// Interpolate to observation location
    void interpolateTL(const LocsL95 &, GomL95 &) const;
    void interpolateAD(const LocsL95 &, const GomL95 &);

    /// Access to data... Could we do without that?
    FieldF90 ** getFields() {return FieldL95::toFortran();}
    const FieldF90 * const * getFields() const {return FieldL95::toFortran();}

protected:
    void initTL(const TLML95 &);
    void initAD(const TLML95 &);
    void stepTL(const TLML95 &, const ModelError &);
    void stepAD(const TLML95 &, ModelError &);

    void accumul(const double & zz, const StateL95 & xx);
};

```

- States are similar but without the linear algebra.
- States and Increments are used by OOPS directly.
- OOPS also adds functionality by defining sub-classes (decorator).

ModelState and ModelIncrement

```
template <typename MODEL> class ModelState: public MODEL::State,
      private util::ObjectCounter<ModelState<MODEL> >
{
    typedef typename MODEL::ModelAuxControl      ModelAuxCtrl_;
    typedef typename MODEL::ModelConfiguration    ModelConfig_;
    typedef typename MODEL::State                 State_;

public:
    ModelState(const ModelConfig_ &, const util::Config &);
    ModelState(const State_ &, const ModelConfig_ &);
    ~ModelState();

    /// Run a forecast
    void forecast(const ModelAuxCtrl_ &, const util::Duration &,
                 PostProcessor<State_> &);

    static const std::string classname() {return "ModelState";}

private:
    const ModelConfig_ & model_;
};
```

- It is a templated class, the template argument is a model trait.
- Note the reference to a ModelConfig object.

ModelState and ModelIncrement

```
template<typename MODEL>
void ModelState<MODEL>::forecast(const ModelAuxCtrl_ & mctl, const util::Duration & len,
                               PostProcessor<State_> & post) {
    const util::DateTime end(validTime() + len);
    LOG(Info) << "ModelState:forecast: Starting forecast, time is " << validTime();
    LOG(Info) << "Start NL" << *this;

    post.initialize(validTime(), end, model_.timestep());
    this->init(model_);
    post.process(*this);

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

    ASSERT(validTime() == end);
    post.finalize();
    LOG(Info) << "ModelState:forecast: Finished forecast, time is " << validTime();
    LOG(Info) << "End NL" << *this;
}
```

- 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 (step) is to move the state in time, nothing else.

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...

Outline

- 1 Complexity
 - Computing complexity
 - Model complexity
 - Data assimilation complexity
- 2 What can we do?
- 3 OOPS design
 - OOPS Design: Abstract Level
 - Implementing the Abstract Design: Building Blocks
 - **Implementing the Abstract Design: Applications**
 - PyOOPS
- 4 From IFS to OOPS

- 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.

Outline

1 Complexity

- Computing complexity
- Model complexity
- Data assimilation complexity

2 What can we do?

3 OOPS design

- OOPS Design: Abstract Level
- Implementing the Abstract Design: Building Blocks
- Implementing the Abstract Design: Applications
- PyOOPS

4 From IFS to OOPS

OOPS Suites and Scripts

- 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 or ecfLOW).
- The three levels could be, and should be, isolated from each other.

Abstracting the workflow

```
dassim = oops4dvar(userConfig)
Bmatrix = mars.retrieve(Bconfig)

for date in daterange(fcycle, lcycle, step):
    obs = mars.retrieve(date, obsConf)
    background = mars.retrieve( fc(date-step, step) )

    an = dassim.run(obs, background, Bmatrix)

    fc = forecast.run(an)

mars.archive(an)
mars.archive(fc)
```

- On its own, the cycling algorithm is relatively easy to describe.
- And there is enough information to **generate** all the triggers!
- Why are we writing them by hand?
 - We are duplicating information.
 - It is difficult to maintain and modify.
 - The risk of bugs is increased.

Prototype: PyOOPS

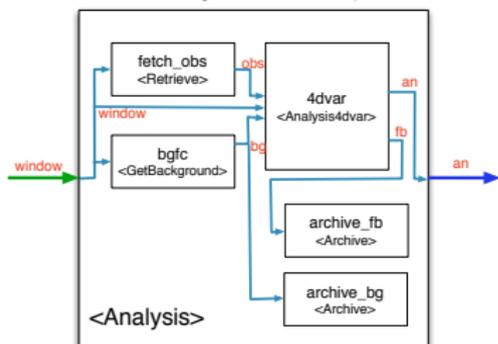
- A prototype has been implemented in python to test the approach.
- The system is organised around **tasks** whose input and outputs are **metadata** objects.
- The metadata objects are also used by the workflow to generate the triggers.

```
class ForecastModel(Task):  
  
    def constructor(self):  
        self.add_input('init')  
        self.add_output('fc')  
        self.add_variable('length')  
        self.add_variable('steps')  
  
    def execute(self):  
        analysis = self.input('init')  
        forecast = MetaData( type = 'fc',  
                             date = analysis.valid_time,  
                             steps = self.variable('steps'),  
                             window_end = analysis.window_end )  
  
        """ code here that configures and executes the model """  
  
        self.set_output('fc', forecast )
```

Prototype: 4D-Var Analysis Cycle

Tasks are used as building blocks to **compose** complex structures

Analysis example



```

class Analysis(CompositeTask):

def constructor(self):
    self.add_input('window')
    self.add_output('an')

    self.fetch_obs = self.add_task( Retrieve('fetch_obs') )
    self.bgfc = self.add_task( GetBackground('bgfc') )
    self.an4dvar = self.add_task( Analysis4dvar('4dvar') )
    self.archive_bg = self.add_task( Archive('archive_bg') )
    self.archive_fb = self.add_task( Archive('archive_fb') )

def compose(self):
    window = self.input('window')

    bg = self.bgfc(window=window)
    obs = self.fetch_obs(window=window)
    (an,fb) = self.an4dvar(bg=bg, obs=obs, window=window)
    self.archive_bg(data=bg)
    self.archive_fb(data=fb)

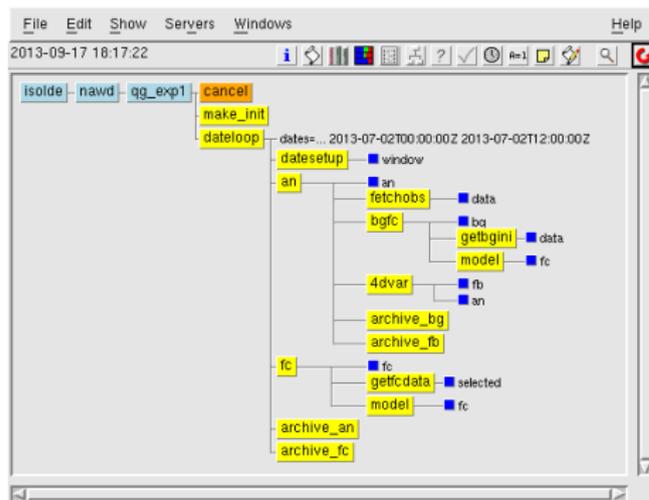
    self.set_output('an', an)

...

datesetup = DateSetup('datesetup')
analysis = Analysis('analysis')

window = datesetup(date='2013-07-02T00:00:00Z')
an = analysis(window=window)
  
```

Prototype with QG toy-model and ecFlow



```
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 GetBackgrounddata 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.

Outline

- 1 Complexity
 - Computing complexity
 - Model complexity
 - Data assimilation complexity
- 2 What can we do?
- 3 OOPS design
 - 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 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.
 - Explore and improve scalability.
 - Changes in one application do not affect other applications.
 - Ability to handle different models opens the door for coupled DA.
- OOPS does not solve scientific problems in itself: it provides a more powerful way to “tell the computer what to do”.
- 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).

- Work to adapt the IFS to OOPS is well under way.
- NEMOVAR will be ported into OOPS.
- OOPS will be available under an open source licence (Apache 2).