Object-Oriented Prediction System

Yannick Trémolet

ECMWF

November 2013

Why OOPS?

- The IFS is a very good global weather forecasting system. However, continuous improvements are necessary to stay at the forefront.
- There is uncertainty in scientific methods that will be used in the future, for example in the data assimilation and dynamical core areas.
- Scalability has become a major concern in view of new computer architectures: addressing it will require significant algorithmic changes.
  - The code can be optimized routine by routine to increase scalability only up to a certain point.
  - Significant leaps in the level of available parallelism can only be achieved through scientific progress in the formulation of the algorithms.
- A very flexible code is needed to test such developments and ideas.
- The code must also be reliable, efficient and readable.

Flexible

- It should be easy to modify the system (new science, new functionality, better scalability...)
- Different concepts should be treated in different parts of the code.
- 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.

Reliable

- The code must run without crashing.
- Additional aspects of reliablity 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.

Reliable

- The code must run without crashing.
- Additional aspects of reliablity 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.

No, a print in a many-Mb-long logfile is not enough!

No, you are not the only one using that variable!

- A controlled abort with a clear error message is not a crash: it saves computer and user time (our time).

Reliable

- The code must run without crashing.
- Additional aspects of reliablity 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.

No, a print in a many-Mb-long logfile is not enough!

No, you are not the only one using that variable!

- 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 (not scientific evaluation),
  - Mecanism to run all the tests easily,
  - Tests run automatically on push to source repository (ECMWF).

Efficient vs. Readable

- The IFS is one of the most computationaly efficient and scalable weather forecasting systems.
- The maintenance cost has become very high and new releases take longer and longer to create and debug.
- It is more and more difficult for newcomers to learn the system and it takes longer to be productive.
- Readable code is not less efficient.
- Readability is staff efficiency: it is as important as computational efficiency (it's just more difficult to measure).

Object-Oriented

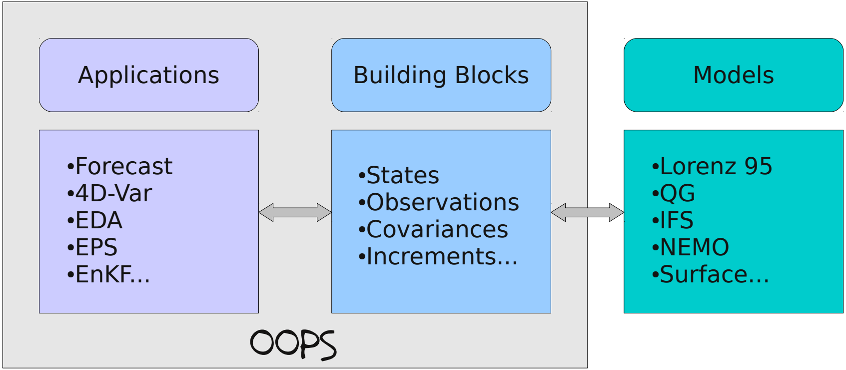
- Flexible, reliable, readable, efficient.
- This is not specific to the IFS: all developers want codes that are modular, reliable, flexible and efficient.
- Since the IFS was designed, in the late 1980's, the software industry has progressed tremendously to make this possible.
- The techniques that have emerged to answer these needs are called **generic** and **object-oriented** programming.
- We have started to re-design our system using this technology in the

  Object-Oriented Prediction System (OOPS).

Object-Oriented and Weather Forecasting

- 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.
- All aspects exist but scientists focus on one aspect at a time: the code should reflect this.
- Object-oriented programming does not solve scientific problems in itself: it provides a more powerful way to tell the computer what to do.
- 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.

What is OOPS?



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

OOPS Implementation

- We have defined a small set of abstract classes that encompasses most entities required for data assimilation.
  - Biases (model and observations) will also be needed.
- For practical implementation, a few more classes will be useful.
- Utility classes:
  - Config, DateTime, Duration, Logger...
- Auxiliary classes:
  - Geometry, ModelConfiguration, TLM (Trajectory), Locations, ModelAtLocations (GOM)

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 ⇓

```
struct QgTraits {
  typedef qg::QgGeometry                  Geometry;
  typedef qg::QgState                     State;
  typedef qg::QgModel                     ModelConfiguration;
  typedef qg::QgIncrement                 Increment;
  typedef qg::QgTLM                       LinearModel;
  typedef oops::NullModelAux              ModelAuxControl;
  typedef oops::NullModelAux              ModelAuxIncrement;

  typedef qg::QgObservation               ObsOperator;
  typedef qg::ObsTrajQG                   ObsOperatorLinearizationTrajectory;
  typedef oops::NullObsAux                ObsAuxControl;
  typedef oops::NullObsAux                ObsAuxIncrement;
  typedef qg::ObsVecQG                    ObsVector;

  typedef qg::LocQG                       Locations;
  typedef qg::GomQG                       ModelAtLocations;

  typedef qg::LocalizationMatrixQG        LocalizationMatrix;
};
```

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

Model Trait Definition

Actual implementation ⇓ Name used in OOPS ⇓

```
struct IfsTraits {
  typedef ifs::GeometryIFS              Geometry;
  typedef ifs::StateIFS                 State;
  typedef ifs::ModelIFS                 ModelConfiguration;
  typedef ifs::IncrementIFS             Increment;
  typedef ifs::LinearModelIFS           LinearModel;
  typedef oops::NullModelAux            ModelAuxControl;
  typedef oops::NullModelAux            ModelAuxIncrement;

  typedef ifs::AllObs                   ObsOperator;
  typedef ifs::AllObsTraj               ObsOperatorLinearizationTrajectory;
  typedef oops::NullObsAux              ObsAuxControl;
  typedef oops::NullObsAux              ObsAuxIncrement;
  typedef ifs::ObsVector                ObsVector;

  typedef ifs::LocationsIFS             Locations;
  typedef ifs::GomsIFS                  ModelAtLocations;

  typedef ifs::LocalizationMatrixIFS    LocalizationMatrix;
};
```

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

Run time vs. Compile time polymorphism

- The model is chosen at compile time via template instantiation.

```
#include "mains/RunQg.h"
#include "model/QgTraits.h"
#include "oops/runs/Forecast.h"

int main(int argc, char ** argv) {
  qg::RunQg< oops::Forecast<qg::QgTraits> > run(argc, argv);
  int info = run.execute();
  return info;
};
```

```
ifs::RunIfs< oops::Forecast<ifs::IfsTraits> > run(argc, argv);
```

- The covariance matrices are chosen at run time because some are generic (Ensemble or hybrid **B**, diagonal **R**).

- The classes in the trait definition might be abstract base classes (see QgObservation).

Encapsulating Fortran Code in C++ Classes

C++

```
Class MyClass {
public:



  doSomething() {
    do_work(&data_);
  }

private:
  Fdata * data_;
}


// Give a class to pointer
Class Fdata {};
```

Interface (ISO)

```
subroutine do_work(c_self)
use iso_c_bindings
use mytype_mod

type(c_ptr) :: c_self
type(mytype), pointer :: self

call c_f_pointer(c_self, self)
call do_it(self)

end subroutine do_work
```

Fortran

```
module mytype_mod

type mytype
! some contents here...
end type mytype

contains



subroutine do_it(self)
type(mytype) :: self
! do the work...
end subroutine do_it

end module mytype_mod
```

Encapsulating Fortran Code in C++ Classes

C++

```
Class MyClass {
public:
  MyClass() {
    create_data(&data_);
  }



  doSomething() {
    do_work(&data_);
  }

private:
  Fdata * data_;
}


// Give a class to pointer
Class Fdata {};
```

Interface (ISO)

```
subroutine create_data(c_self)
use iso_c_bindings
use mytype_mod

type(c_ptr) :: c_self
type(mytype), pointer :: self

allocate(self)
call create(self)
c_self = c_loc(self)

end subroutine create_data
```

Fortran

```
module mytype_mod

type mytype
! some contents here...
end type mytype

contains

subroutine create(self)
type(mytype) :: self
! allocate and setup...
end subroutine create


subroutine do_it(self)
type(mytype) :: self
! do the work...
end subroutine do_it

end module mytype_mod
```

- No static variable of type `mytype` is declared in the module!

Encapsulating Fortran Code in C++ Classes

C++

```
Class MyClass {
public:
  MyClass() {
    create_data(&data_);
  }

  ~Myclass() {
    delete_data(&data_);
  }

  doSomething() {
    do_work(&data_);
  }

private:
  Fdata * data_;
}


// Give a class to pointer
Class Fdata {};
```

Interface (ISO)

```
subroutine do_work(c_self)
use iso_c_bindings
use mytype_mod

type(c_ptr) :: c_self
type(mytype), pointer :: self

call c_f_pointer(c_self, self)
call do_it(self)

end subroutine do_work
```

Fortran

```
module mytype_mod

type mytype
! some contents here...
end type mytype

contains

subroutine create(self)
type(mytype) :: self
! allocate and setup...
end subroutine create

subroutine delete(self)
type(mytype) :: self
! deallocate...
end subroutine delete

subroutine do_it(self)
type(mytype) :: self
! do the work...
end subroutine do_it

end module mytype_mod
```

- No static variable of type `mytype` is declared in the module!
- The Fortran module does not know about C++: it is fully usable in the rest of the Fortran code.

Object-Oriented Prediction System

Yannick Trémolet

ECMWF

November 2013

Getting OOPS

- The main point of information about OOPS is the wiki page:
  `https://software.ecmwf.int/wiki/display/OOPS/OOPS+Home`
- The source code is accessible from the git repository (via stash):
  `https://software.ecmwf.int/stash/projects/OOPS`
- The IFS component are in the usual perforce repository.

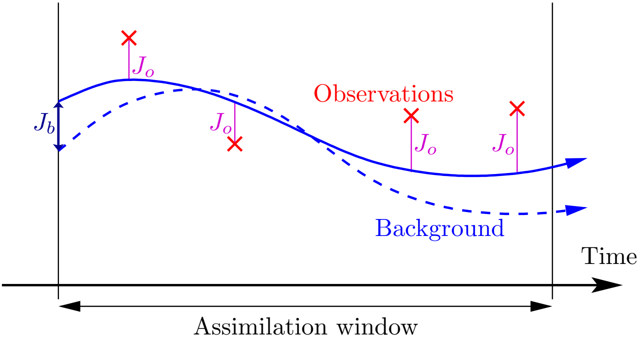
OOPS Design

- Why OOPS?
  - What do we want to develop?
  - Why cannot we do it in the IFS?
- OOPS General Design
  - How can we adress the problems above?
  - What basic classes do we need (building blocks)?
  - Run time vs. compile time polymorphism
- Details of some classes
  - Basic classes: State, Observations
  - Building a DA system: CostFunction, Minimizer
- Not enough time to cover every class in OOPS
  - Enough to understand the main structure
  - Examples of "object-thinking"

Outline

1. The IFS
2. OOPS Design: Abstract Level
3. Implementing the Abstract Design: Building Blocks
4. Implementing the Abstract Design: Applications
5. Some General Comments

The IFS was designed for 4D-Var

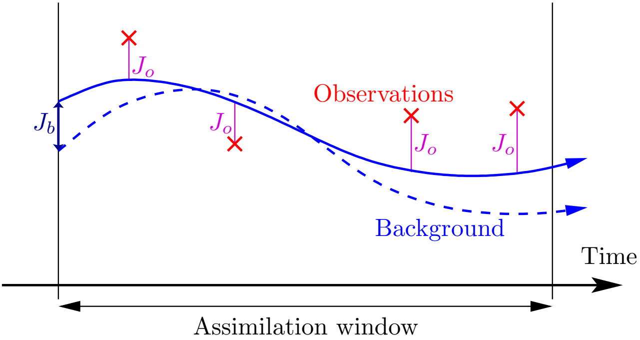


- The initial state is integrated forward and compared with the observations.
- The 4D-Var cost function is computed

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

- and minimized using an incremental approach.

The IFS was designed for 4D-Var

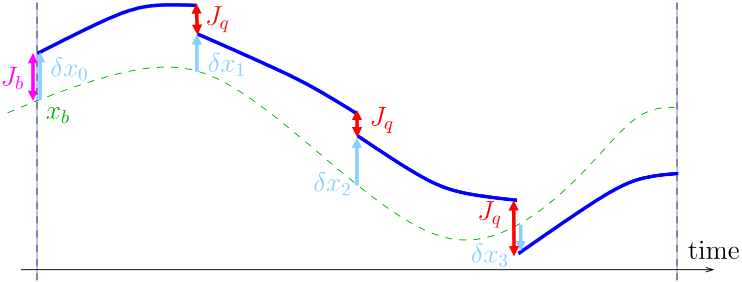


- The initial state is integrated forward and compared with the observations.
- The 4D-Var cost function is computed

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

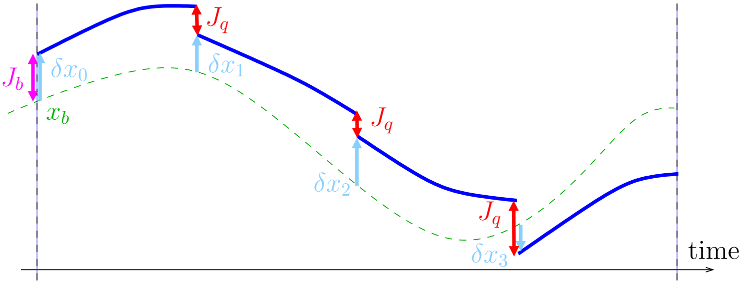
- and minimized using an incremental approach.

Weak Constraint 4D-Var



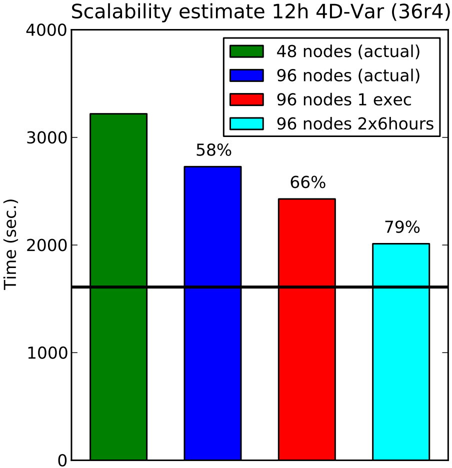
- The control variable is the state at several points in time.
- There are additional terms in the cost function.
- Model integrations over each sub-window are independent.

Weak Constraint 4D-Var



- The control variable is the state at several points in time.
- There are additional terms in the cost function.
- Model integrations over each sub-window are independent.
- We need several states!
- The nature of the optimization problem is different: we need to explore dual space (i.e. observation space) algorithms (or mixed primal/dual).

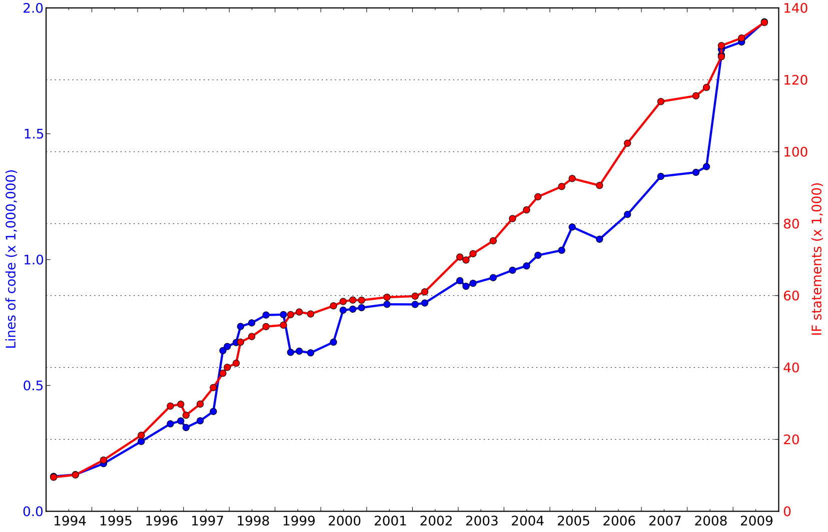
More Scalability in 4D-Var



Estimates from Deborah Salmond

- Incremental 4D-Var in the IFS is achieved by executing the IFS several times.
- Running 1 executable instead of 7 would reduce I/O and start-up costs.
- We need states and increments at different resolutions (inner and outer loops).

Another concern: IFS complexity



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

Current situation in the IFS

- Most high level routines don't have arguments (global variables).
  - Assumes that there is only one state, one set of observations, one...
  - 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 practically 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.
  - It is not possible to run 4D-Var in one executable which affects performance.
- In practice, only one resolution can be used per execution.

Outline

1. The IFS
2. OOPS Design: Abstract Level
3. Implementing the Abstract Design: Building Blocks
4. Implementing the Abstract Design: Applications
5. Some General Comments

- 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.

- 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.

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.

- 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.

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

- 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

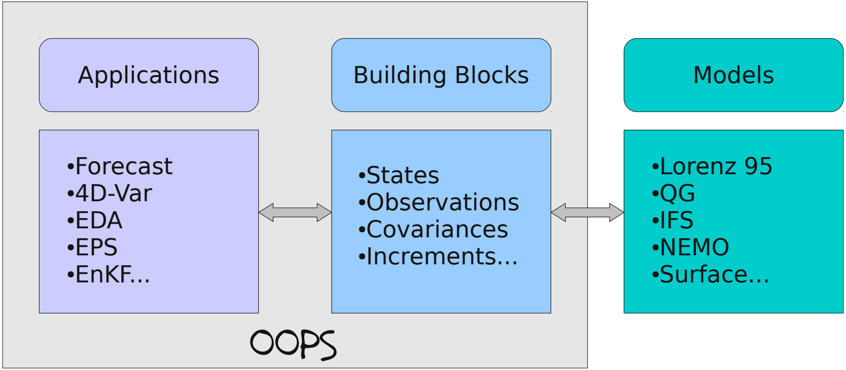
$$J(\mathbf{x}) = \frac{1}{2}(\mathbf{x}_0 - \mathbf{x}_b)^T \mathbf{B}^{-1}(\mathbf{x}_0 - \mathbf{x}_b) + \frac{1}{2}\sum_{i=0}^{n}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]^T \mathbf{R}_i^{-1}[\mathcal{H}(\mathbf{x}_i) - \mathbf{y}_i]$$

- 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 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.

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

1 The IFS

2 OOPS Design: Abstract Level

3 Implementing the Abstract Design: Building Blocks

4 Implementing the Abstract Design: Applications

5 Some General Comments

OOPS Implementation

- We have defined a small set of abstract classes that encompasses most entities required for data assimilation.
  - Biases (model and observations) will also be needed.
- For practical implementation, a few more classes will be useful.
- Utility classes:
  - Config, DateTime, Duration, Logger...
- Auxiliary classes:
  - Geometry, ModelConfiguration, TLM (Trajectory), Locations, ModelAtLocations (GOM)

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 ⇓

```
struct QgTraits {
  typedef qg::QgGeometry                Geometry;
  typedef qg::QgState                   State;
  typedef qg::QgModel                   ModelConfiguration;
  typedef qg::QgIncrement               Increment;
  typedef qg::QgTLM                     LinearModel;
  typedef oops::NullModelAux            ModelAuxControl;
  typedef oops::NullModelAux            ModelAuxIncrement;

  typedef qg::QgObservation             ObsOperator;
  typedef qg::ObsTrajQG                 ObsOperatorLinearizationTrajectory;
  typedef oops::NullObsAux              ObsAuxControl;
  typedef oops::NullObsAux              ObsAuxIncrement;
  typedef qg::ObsVecQG                  ObsVector;

  typedef qg::LocQG                     Locations;
  typedef qg::GomQG                     ModelAtLocations;

  typedef qg::LocalizationMatrixQG      LocalizationMatrix;
};
```

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

Model Trait Definition

Actual implementation ⇓   Name used in OOPS ⇓

```
struct IfsTraits {
  typedef ifs::GeometryIFS              Geometry;
  typedef ifs::StateIFS                 State;
  typedef ifs::ModelIFS                 ModelConfiguration;
  typedef ifs::IncrementIFS             Increment;
  typedef ifs::LinearModelIFS           LinearModel;
  typedef oops::NullModelAux            ModelAuxControl;
  typedef oops::NullModelAux            ModelAuxIncrement;

  typedef ifs::AllObs                   ObsOperator;
  typedef ifs::AllObsTraj               ObsOperatorLinearizationTrajectory;
  typedef oops::NullObsAux              ObsAuxControl;
  typedef oops::NullObsAux              ObsAuxIncrement;
  typedef ifs::ObsVector                ObsVector;

  typedef ifs::LocationsIFS             Locations;
  typedef ifs::GomsIFS                  ModelAtLocations;

  typedef ifs::LocalizationMatrixIFS   LocalizationMatrix;
};
```

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

Run time vs. Compile time polymorphism

- The model is chosen at compile time via template instantiation.

```
#include "mains/RunQg.h"
#include "model/QgTraits.h"
#include "oops/runs/Forecast.h"

int main(int argc, char ** argv) {
  qg::RunQg< oops::Forecast<qg::QgTraits> > run(argc, argv);
  int info = run.execute();
  return info;
};
```

```
ifs::RunIfs< oops::Forecast<ifs::IfsTraits> > run(argc, argv);
```

- The covariance matrices are chosen at run time because some are generic (Ensemble or hybrid **B**, diagonal **R**).
- The classes in the trait definition might be abstract base classes (see QgObservation).

Source code

- Top level scientific code in `src/oops/runs`
- The structure for oops source code is:

```
/path/to/oops
├── src
│   ├── logger
│   ├── oops
│   │   ├── assimilation
│   │   ├── base
│   │   └── runs
│   ├── test
│   └── util
├── l95
├── qg
└── ...
```

Forecast class

```
namespace oops {

template <typename MODEL> class Forecast {
  typedef typename MODEL::Geometry              Geometry_;
  typedef typename MODEL::ModelAuxControl       ModelAuxCtrl_;
  typedef typename MODEL::ModelConfiguration    ModelConfig_;
  typedef typename MODEL::State                 State_;

 public:
  Forecast() {}
  ~Forecast() {}
  int execute(const util::Config &)
};

}  // namespace oops
```

- The typedefs are aliases to shorter names to avoid repeating the entire name: `typename MODEL::Geometry`
- `MODEL::Geometry` would have been nicer but in many places it is not enough...
- The short names are consistent throughout the code (generated by script).

Forecast class

```
  int execute(const util::Config & fullConfig) {
//  Setup resolution
    const util::Config resolConfig(fullConfig, "resolution");
    const Geometry_ resol(resolConfig);

//  Setup ModelConfig_
    const util::Config modelConfig(fullConfig, "model");
    const ModelConfig_ model(resol, modelConfig);

//  Setup initial state
    const util::Config initialConfig(fullConfig, "initial");
    LOG(Configs) << "Initial configuration is:\n" << initialConfig;
    ModelState<MODEL> xx(model, initialConfig);
    LOGS(Info, Test) << "Initial state:" << xx;

//  Setup augmented state
    ModelAuxCtrl_ moderr(initialConfig);

//  Setup times
    const util::Duration fclength(fullConfig.getData("forecast_length"));
    const util::DateTime bgndate(xx.validTime());
    const util::DateTime enddate(bgndate + fclength);
    LOG(Info) << "Running forecast from " << bgndate << " to " << enddate;

//  Setup forecast outputs
    PostProcessor<State_> post;
    const util::Config outConfig(fullConfig, "output");
    post.enrollProcessor(new StateWriter<State_>(bgndate, outConfig));

//  Run forecast
    xx.forecast(moderr, fclength, post);

    LOGS(Info, Test) << "Final state:" << xx;
    return 0;
  }
```

A simple class: Geometry (L95)

```
class Resolution {
 public:
  explicit Resolution(const util::Config & conf): resol_(conf.getInt("resol")) -
  Resolution(const Resolution & other): resol_(other.resol_) {}
  ~Resolution() {}

  int toFortran() const {return resol_;}

  friend std::ostream & operator<< (std::ostream &, const Resolution &);

 private:
  const int resol_;
};
```

- OOPS expects very little from such a class.
- Some method are specific and not used by OOPS (`toFortran`).

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::DateTime
  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.

Observations

```
template <typename MODEL> class Observations {
 public:
  Observations(const util::Config &, const util::DateTime &, const util::DateTime &);
  Observations(const Observations &, const bool copyObs = true);
  ~Observations();
  Observations & operator=(const Observations &);

/// Interactions with Departures
  Departures_ * newDepartures(const std::string & name = "") const;
  Departures_ operator-(const Observations & other) const;
  Observations & operator+=(const Departures_ &);

/// Get GOM
  GOM_ * newGOM(const util::DateTime &, const util::DateTime &) const;

/// Get observations locations
  ObsLocations_ * locations(const util::DateTime &, const util::DateTime &) const;

/// Get observation operator trajectory
  ObsOpTraj_ * newObsTraj() const;

/// Compute observations equivalents
  void runObsOperator(const GOM_ &, const ObsAuxCtrl_ &);
  void runObsOperator(const GOM_ &, const ObsAuxCtrl_ &, ObsOpTraj_ &);

/// Save observations values
  void save(const std::string &) const;

/// Print human readable observations informations
  friend std::ostream & operator<< <> (std::ostream &, const Observations &);
```

Observations

```
template <typename MODEL> class Observations {
 public:
// continued...

/// Generate observation distribution
  void generateDistribution(const util::Config &);

/// Start of assimilation window
  const util::DateTime & windowStart() const {return winbgn_;}
/// End of assimilation window
  const util::DateTime & windowEnd() const {return winend_;}

/// Double despatch for obs error covariance methods
  ObsErrorBase_ * helpCovarCreate(const util::Config & conf) const;
  void helpCovarLinearize(ObsErrorBase_ & R) const {R.linearize(*obs_);}

 private:
  boost::shared_ptr<ObsOperator_> oper_;
  boost::scoped_ptr<ObsVector_> obs_;
  const util::DateTime winbgn_;
  const util::DateTime winend_;
};
```

- The double despatch is a technique to preserve the encapsulation.
- The smart pointers take care of the memory mangement for us.

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. The IFS
2. OOPS Design: Abstract Level
3. Implementing the Abstract Design: Building Blocks
4. Implementing the Abstract Design: Applications
5. Some General Comments

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.
- The terms can be re-used (or not) for various formulations (3D-Var, 4D-Var, weak constraint, 4D-Ens-Var...).
- 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 choice of cost function is made at run time via a factory.

Cost Function Base Class

```
template<typename MODEL> class CostFunction : private boost::noncopyable {
 public:
  explicit CostFunction(const ModelConfig_ &);
  virtual ~CostFunction() {}

  double evaluate(const CtrlVar_ &, const util::Config &, PostProcessor<State_> &) const;
  double linearize(const CtrlVar_ &, const util::Config &, PostProcessor<State_> &);

  virtual void runTLM(CtrlInc_ &,
                      PostProcessor<Increment_>&, PostProcessorTL<Increment_>&) const =0;
  virtual void runADJ(CtrlInc_ &,
                      PostProcessor<Increment_>&, PostProcessorAD<Increment_>&) const =0;

  void addIncrement(CtrlVar_ &, const CtrlInc_ &,
                    PostProcessor<Increment_> post = PostProcessor<Increment_>() ) const;
  void resetLinearization();

  /// Access to Jb and other terms of the cost function
  const CostJbBase<MODEL> & jb() const {return *jb_;}
  const CostBase_ & jterm(const unsigned ii) const {return jterms_[ii];}

  /// Cost function gradient at first guess.
  CtrlInc_ getGradientFG() const;

 private:
  virtual void runNL(CtrlVar_ &, PostProcessor<State_>&) const =0;

// Data members
  const ModelConfig_ & model_;
  boost::scoped_ptr<CostJbBase<MODEL> > jb_;
  boost::ptr_vector<CostBase_> jterms_;
  boost::ptr_vector<LinearModel_> tlm_;
};
```

A few methods have been removed for the presentation.

Cost Function Base Class

```
template<typename MODEL>
double CostFunction<MODEL>::evaluate(const CtrlVar_ & fguess,
                                     const util::Config & config,
                                     PostProcessor<State_> & post) const {
// Setup terms of cost function
  PostProcessor<State_> pp(post);
  for (unsigned jj = 0; jj < jterms_.size(); ++jj) {
    pp.enrollProcessor(jterms_[jj].initialize(fguess));
  }

// Run NL model
  CtrlVar_ xx(fguess);
  this->runNL(xx, pp);

// Cost function value
  const util::Config diagnostic(config, "diagnostics", true);
  double zzz = jb_->evaluate(fguess, xx);
  for (unsigned jj = 0; jj < jterms_.size(); ++jj) {
    zzz += jterms_[jj].finalize(diagnostic);
  }
  LOGS(Info, Test) << "CostFunction: Nonlinear J = " << zzz;
  return zzz;
}
```

- Saving the model linearization trajectory is also the responsibility of a PostProcessor.
- Only J_b has a special role.

4D-Var Cost Function

```
template<typename MODEL> class CostFct4DVar : public CostFunction<MODEL> {
 public:
  CostFct4DVar(const util::Config &, const ModelConfig_ &);
  ~CostFct4DVar() {}

  void runTLM(CtrlInc_ &,
              PostProcessor<Increment_>&, PostProcessorTL<Increment_>&) const;
  void runADJ(CtrlInc_ &,
              PostProcessor<Increment_>&, PostProcessorAD<Increment_>&) const;

 private:
  void runNL(CtrlVar_ &, PostProcessor<State_>&) const;
  void addIncr(CtrlVar_ &, const CtrlInc_ &, PostProcessor<Increment_>&) const;

  CostJb<MODEL>       * newJb(const util::Config &, const Geometry_ &) const;
  CostJo<MODEL>       * newJo(const util::Config &) const;
  CostTermBase<MODEL> * newJc(const util::Config &, const Geometry_ &) const;

  util::Duration windowLength_;
  util::DateTime windowBegin_;
  util::DateTime windowEnd_;
};
```

- Specific implementations of abstract methods from the base class.

Using the Cost Function

- The cost function is created by a factory according to the configuration (`src/oops/runs/Variational.h`)

```
//  Setup cost function
  const util::Config cfConf(fullConfig, "cost_function");
  boost::scoped_ptr< CostFunction<MODEL> >
      J(CostFactory<MODEL>::create(cfConf, model));
```

  - A smart pointer is used to take ownership of the pointer returned by the factory...
  - because the factory cannot return a reference.

- It is very easy to add new cost function implementations.

- 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. The IFS
2. OOPS Design: Abstract Level
3. Implementing the Abstract Design: Building Blocks
4. Implementing the Abstract Design: Applications
5. Some General Comments

General Comments about OO Design

- Who owns each object?
- Who has responsibility for a given piece of information? For a given action?
- One class == one responsibility
  - Don't do too much but do it well.
  - Focus on one problem at a time.
  - Compose classes for more complex tasks.
- Use of smart pointers:
  - When possible use a reference,
  - If a reference won't do, try a scoped pointer,
  - If a scoped pointer doesn't work, try a shared pointer,
  - If a shared pointer doesn't work, try an auto (or unique) pointer,
  - Only if all else fails use a plain pointer (risk of memory leak, lacks information about the intent of the design).