Contents

I Objectives 3

1 Introduction 3

2 Contents of the Library 3
  2.1 Assimilation Methods 3
    2.1.1 Basic Methods 3
    2.1.2 Kalman Filters 3
    2.1.3 Variational Methods 4
    2.1.4 Ensemble Forecast with Sequential Aggregation 4
    2.1.5 Other Methods 4
    2.1.6 Note 4
  2.2 Tools 4
    2.2.1 Observations 4
    2.2.2 Models 5
    2.2.3 Error Statistics 5
    2.2.4 Other Tools 5

II Constraints 5

3 Contents 5

4 Computer Languages 6
  4.1 A Low-Level Language 6
  4.2 A High-Level Language 6
Part I

Objectives

1 Introduction

The leading idea is to develop a data assimilation library intended to be generic, at least for high-dimensional systems. Further explanations about the objectives of the library are given in the appendix B (in French).

2 Contents of the Library

This section describes the contents of the library as it should be seen from the user’s viewpoint. Technical contents (compilation facilities, unit tests, ...) are listed in section 9.

2.1 Assimilation Methods

2.1.1 Basic Methods

1. Nudging, back and forth nudging [Auroux and Blum, 2005]

2. Optimal interpolation

2.1.2 Kalman Filters

3. Kalman filter


5. Low-rank Kalman filters [PSF and PEF in Cohn and Todling, 1996], also the reduced-rank square root (RRSQRT) Kalman filter [Verlaan and Heemink, 1995]

6. Mixed ensemble and RRSQRT Kalman filter [Heemink et al., 2001]

7. Singular evolutive extended Kalman (SEEK) [Pham et al., 1996]

8. Singular evolutive interpolated Kalman (SEIK) [Pham, 2001]

2.1.3 Variational Methods

10. 3D-Var
11. 4D-Var
12. Incremental 4D-Var?

2.1.4 Ensemble Forecast with Sequential Aggregation

14. Machine learning methods [Cesa-Bianchi and Lugosi, 2006]: gradient descent, exponentiated gradient, ...
15. Statistical methods: dynamic linear regression [West and Harrison, 1999], ...

2.1.5 Other Methods

16. At least one particle filter [Gordon et al., 1993; Doucet et al., 2001]
17. Methods based on the principle of maximum entropy on the mean [Golan and Gzyl, 2002; Bocquet, 2005]

2.1.6 Note

Whenever relevant, the implementation should support parameter estimation.

2.2 Tools

2.2.1 Observations

1. Template observation operator

2. Observation operator mapping from gridded data to pointwise data (observations taken at a list of locations, like in a network), at least for 1D, 2D and 3D grids (or maybe for any dimension)

3. Generation of synthetic observations (at a list of locations, for instance) based on a model, possibly with perturbations
2.2.2 Models

4. Template model
5. Sample models: shallow water, Lorenz models
6. Generic tangent linear model based on finite differences
7. Validation of tangent linear models and adjoint models

2.2.3 Error Statistics

8. Classical error covariance matrices: diagonal, Gaussian form, Balgovind form [Balgovind et al., 1983], see also Gaspari and Cohn [1999]
9. Covariance localization and inflation
10. $\chi^2$ diagnosis [e.g., Ménard et al., 2000]
11. Perturbation of a multidimensional field on the basis of given error statistics (with space and time dependence)

2.2.4 Other Tools

12. Interfaces to useful minimization algorithms: conjugate gradient method, BFGS [Byrd et al., 1995], ...
13. Twin experiments (this is related to the generation of synthetic observations)
14. Performance measures: correlation, root mean square error, bias, fractional bias, ...

Part II

Constraints

3 Contents

As mentioned in section 1, the library should be designed for high-dimensional problems first. This does not mean that other problems cannot be addressed, but if the high dimension demands a specific design in some respect, there should be no concern about following that design.
4 Computer Languages

4.1 A Low-Level Language

Obviously there must be a low-level language since high performance is required. This language must be

1. efficient for computing;
2. with rich features, especially when it comes to genericity;
3. coming along with libraries that cover the needs of the present library (primarily linear algebra).

4.2 A High-Level Language

In scientific computing, one always relies at some point on a high-level language. This language is used at least for visualization. The library should communicate well with one or more high-level languages. This should permit direct calls to the library, so that one gets at hand the results of the assimilation, and maybe the intermediate steps if the interface to the methods is rich enough.

In this context, a high-level language should offer:

1. the ability to make calls to many methods and functions of the library;
2. the ability to manipulate the library data structures (so as to access the library results);
3. the possibility to easily generate the interface to the library (more or less automatically so that any change in the library should not result in significant workload);
4. no significant loss of performance.

The use of the high-level language should always remain optional. The library should work without it.

5 Implementation

5.1 Portability

It is needless to stress the demand for portability since it is a natural feature of any good software. The library should compile on BSD systems, Linux, MacOS, Unix and Windows. Beyond the portability itself, this often ensures that most compilers will accept the library, and this may even increase the safety of the implementation.
An obvious consequence is that all dependencies of the library must be portable.

Portability does not mean poor software because of constraints it would put on the implementation. Portability rather means following the languages standards. In case a platform cannot offer a decent compiler or interpreter (i.e., a compiler or an interpreter that understands standard-compliant code), the platform can be discarded—and probably should.

5.2 Long Term

The design, the day-to-day choices, the implementation should remain compatible with long-term development, and they should even enforce the long-term objectives. One consequence is not to rush in the implementation without a careful design. Another is that re-writing parts of the library, because their very design is flawed, is perfectly acceptable.

The design should ensure that the library is scalable.

5.3 Maintenance

It is important to lower as much as possible the maintenance costs. This obviously increases the resources for the new developments. And, more important, this enables to widely extend the library without concerns about its viability.

This also contributes to make the library perennial. One does not know what (development) resources will be available in the future. The library should not collapse simply because no engineer is there to maintain it.

5.4 Parallel Computing

Because of the high-dimensional problems to be addressed, parallel computing may be needed. The library should therefore be available for supercomputers and clusters. Nowadays a lot of parallel computing involves multiple cores and processors in the same machine: obviously, this must be supported.

5.5 Compatibility with Users’ Models and Observations

The library should ease plugging a user’s model. It is likely that the model requires adjustments to be compatible with the library, but they should be minimized. The library should rely on the lightest interface and it should assume very little about the model structure.

These constraints relate to genericity. The library should be able to deal with many applications, thus many models, observations and error statistics.
5.6 Nesting and Coupling

The assimilation is not necessary the last stage in a system. The model with assimilation may be seen as a new model which is in turn included in something. For instance, a model with assimilation, say a meteorological model, may be coupled with another model, like a surface model. Hence assimilation should never be seen as the end of the simulation process.

6 Support

As an open-source code, intended to be shared at some point (the sooner, the better), the library should come with all common documentation: user’s guide, reference documentation, a few examples and a website (even with a single page). It is also a good idea to distribute development-oriented documentation like a coding standard and an explanation of the overall architecture. Sharing is certainly the best way to welcome new users and new contributors in the long run. One might even consider opening the code repository itself.

Part III
Design and Development Process

7 Introduction

A list of software directly related to data assimilation is shown in appendix A.

8 Hints on the Design

Another document will settle the library design. This section merely collects some ideas about a possible design.

8.1 Modular Programming

Long-term development (section 5.2) and modular programming are friends. So the library should be quite modular. This has other advantages, among which:

1. if one part of the library is badly implemented, this has little consequences elsewhere;
2. the library can be extended at low cost since one does not need to master the whole library to contribute locally (e.g., with a new observation manager);

3. maintenance is local; it can therefore be distributed among the contributors, and it should be easier than global maintenance.

It is noteworthy that open-source software is always very modular. It is due to the very nature of the development: many developers are involved, sometimes with limited contributions. It turns out that modular programming is also beneficial to more conventional development.

The different parts of a modular software communicate with predefined interfaces. What is performed behind an interface is the responsibility of the developers involved there.

As René Descartes appropriately wrote, “there is seldom so much perfection in works composed of many separate parts, upon which different hands had been employed, as in those completed by a single master.”\(^1\) Paradoxically, modular programming partially addresses this issue because each module has a limited number of contributors. At the same time, the partitioning may indeed be harmful when the modules start not to be consistent in their internal design and their implementation. This is where a coding standard is important, and where one or more developers should enforce the overall consistency.

### 8.2 Languages

The low-level language will be C++.

It is the most powerful low-level language that can be used for scientific computing.

Nowadays many new projects in the applied-mathematics community are implemented in C++, especially the large projects. Beyond this community, C++ is one of the most used computer languages (surely in top 5, maybe in top 3). It therefore enjoys many tools and the community has lot of support to offer.

Thinking about the long term, C++ is a reasonable choice. Students in engineering schools often learn C++: object-oriented programming may essentially be learned with C++, Java or maybe Python/Ruby. To deal with high-performance computing, there is only one sensible choice among these four languages. In addition, there is no other major language that can currently exhibit the same performance and the same scope of capabilities as C++ and its resources do.

\(^1\) *Discourse on the Method*, 1637. Original quote, in French: “souvent il n’y a pas tant de perfection dans les ouvrages composés de plusieurs pièces, et faits de la main de divers maîtres, qu’en ceux auxquels un seul a travaillé”, *Discours de la méthode*.
Two high-level languages will be used: Python and Matlab.

Matlab has long been used and is shipped with many advanced toolboxes. It also includes a powerful graphical user interface. Its main drawback is that it is not free. Therefore it is seldom available on one’s laptop or at home. Not every institute, laboratory, university or firm has a convenient access to Matlab. Another drawback is that the language is not very powerful (no introspection, no name spaces, ...).

Python is a more powerful language. It is free, its interpreter is free software and many libraries (the so-called modules) are free software too. A very convenient software is SWIG (http://www.swig.org/) which can (almost) automatically build a Python interface to a C/C++ code. The main limitation with Python is that it may not offer enough (complete) modules to replace all Matlab toolboxes. In the recent past, Python has been maturing for scientific computing. It is now pretty well organized around SciPy (http://scipy.org/), NumPy (http://numpy.scipy.org/), Matplotlib (http://matplotlib.sourceforge.net/) and MayaVi/PyVTK (http://mayavi.sourceforge.net/). Some packages try to provide all Python facilities in a single consistent package, like SAGE (http://www.sagemath.org/). Many efforts were explicitly undertaken in order to replace Matlab.

8.3 Views on the Design

A key point is to determine the scope of the classes. In object-oriented programming, people tend to take two extremes stands on the issue: one is to advocate the use of very few classes (therefore large classes), the other is to define as many small classes as possible. The former option leads to a simple library interface and a rather simple overall design for the developers. The latter option gives more flexibility and capabilities, and it may ease the maintenance.

In practice, it should be recommended to define classes that have a natural scope, large or not. For instance, a significant operator, that appears in the equations, should probably be implemented in a single class.

In data assimilation, one can identify:

1. a numerical model;
2. the observations;
3. error statistics, related to the model or to the observations;
4. data in miscellaneous forms: time series, maps, multivariate fields, .....

It seems reasonable to build a class (or several classes with a similar interface) for these four entities. There may also be a class in charge of outputting the results (on storage media).
A data assimilation method itself essentially manipulates matrices and vectors. In order to retrieve these matrices and vectors, and in order to process them, the implementation of the method makes calls to the model and to an observation manager. A key point is therefore the interface that the objects show. Several ideas about what the interfaces may be are provided below.

The following sections propose a rough overview of what the classes may contain. Their eventual contents and, of course, the eventual names may drastically change in the final design. The rough design below is surely not settled. It may be useful as a starting point.

8.3.1 Numerical Model

The required interface will greatly vary with the data assimilation method. The class Model may give access to

1. the time integration over one time step, say with Forward;
2. the state vector, say with GetState;
3. the date that the model has reached (in its time integration), with GetDate that probably returns an integer;
4. the covariance error “models”, say GetStateErrorVariance for background and GetModelErrorVariance for model error, which are objects (see section 8.3.3);
5. the model input data and parameters, so that they could be perturbed;
6. the tangent linear model;
7. the adjoint model, with a backward mode (Backward) and access to adjoint variables;
8. some source term of the underlying equation (for nudging).

8.3.2 Observations

The observations should be handled by a class called observation manager. The interface of ObservationManager may give access to

1. a method that tells the manager to load/read the observations at a given date (integer Model::GetDate);
2. a method that tells whether observations are available at current date, or whether observations have been available between the previous date and the current date, say HasObservation;
3. the observations, with GetData;
4. the covariance error “model”, say \texttt{GetErrorVariance} for the observations;

5. a method to perturb the observations (useful in twin experiments);

6. the tangent linear version;

7. the adjoint version;

8. probably the observation operator.

The observation operator may be implemented in a dedicated class, as the error statistics. It may or may not be encapsulated in the observation manager. The observation operator will need access to the model and to the observation manager, since it maps from model space to observation space. Obviously this demands some mechanism to specify the position of the observations and the positions associated with the model state components (note that parts of the state vector may not be associated with any position).

### 8.3.3 Error Statistics

Many assimilation methods require statistics for observation errors, model errors and/or background errors. The interface of a class for error statistics could include:

1. a method to retrieve the covariance matrix—the method should not be called in high-dimension, though!

2. a method to retrieve a row (or a column) of the covariance matrix;

3. base methods to get the size of the covariance matrix and to access a single element of the matrix—one should be able to use the class as if it was a matrix class (so, consistency with the library for linear algebra would be a good idea).

### 8.3.4 Data Management

There may be a need for several \textit{data containers}. This probably not necessary for the state vector or the observations at a given date, since a vector should be good enough. But if ones manipulates, say, the sequence (in time) of all observations from a network, the data structure is more complex. For instance, at many dates, there may be missing observations at a few locations of the network. In addition, one may need specific methods to remove a location, to apply some filter, to threshold the data, . . .

For sequential aggregation (ensemble forecast), one may rely on an \textit{ensemble container} that stores all simulated values and maybe the corresponding observed values.
More generally, data processing can be painful. A few dedicated classes would be welcome.

8.3.5 Examples

Assume that the model is the object `model` (an instance of class `Model`) and that the observation manager is `observation_manager`.

A simulation without assimilation could be implemented this way:

```c
model.Init();
for (int h = 0; h < model.GetNstep(); h++)
    model.Forward();
```

With sequential assimilation, the implementation would roughly follow this:

```c
model.Init();
observation_manager.Init(model);
for (int h = 0; h < model.GetNstep(); h++)
{
    model.InitStep();
    model.Forward();
    observation_manager.SetDate(model.GetDate());
    if (observation_manager.HasObservation())
    {
        innovation = observation_manager.GetData()
                     - observation_manager.TangentLinearModel(model.GetState());
        // Assuming the gain matrix is K.
        model.GetState() += K * innovation;
    }
}
```

A 4D-Var assimilation would (very) roughly look like the following code (which contains errors to simplify the algorithm, so that one can focus on the overall implementation):

```c
model.Init();
trajectory.Init(model);
observation_manager.Init(model);
// 'Niter' is a fixed number of iterations of the optimization algorithm.
for (int i = 0; i < Niter; i++)
{
    /*** Forward loop ***/
    // Initializes the cost with the background term.
    cost = ...;
    for (int h = 0; h < model.GetNstep(); h++)
    {
        model.InitStep();
        model.Forward();
        trajectory.SetValue(model.GetDate(), model.GetState());
        observation_manager.SetDate(model.GetDate());
        if (observation_manager.HasObservation())
        {
            innovation = observation_manager.GetData()
                        - observation_manager.TangentLinearModel(model.GetState());
            // Assuming the gain matrix is K.
            model.GetState() += K * innovation;
        }
    }
    // Updating the cost after assimilation.
    cost = ...
}
```
- observation_manager.TangentLinearModel(model.GetState());
  cost += ...;
}
}

/*** Backward loop ***/
model.GetAdjointState() = 0.;
for (int h = model.GetNstep(); h >= 1; h--)
{
  model.GetState() = trajectory.GetValue(model.GetDate());
  observation_manager.SetDate(model.GetDate());
  if (observation_manager.HasObservation())
    model.GetAdjointState() += ...
  model.InitBackwardStep();
  model.Backward();
}
// Adds the contribution of the background term to the adjoint state.
model.GetAdjointState() += ...
// Computes the new initial state (optimization algorithm).
model.GetState() = ...;

9 Development Model and Tools

9.1 Development Model

The library will be free software, under GNU GPL. Individual BSD or GNU
LGPL licenses may be granted after request.

A contribution can be committed to the repository if and only if it is (1)
compliant with the overall design, (2) compliant with the coding standard,
(3) of some interest for the library (i.e., not too specific), (4) documented
(or soon documented: stable versions will only include documented code).
This means that a certain quality level must be achieved for a contribution
to be included, but also that, once this quality level is “cleared”, there are
few constraints. This is made possible thanks to the modularity and the
scalability of the library and thanks to the light maintenance that should
be required.

If possible, the developers should write ready-to-commit code and com-
mit themselves. In practice, a new developer should be first supervised,
preferably by an engineer who can detect if a contribution is good enough
for inclusion in the library. There should be a learning period during which
the new developer has her/his contributions (commit candidates) checked
and maybe corrected. After the developer has produced several quality con-
tributions (with no correction needed), say about 4 in a raw, she/he gains
commit rights.

There should always be someone (preferably an engineer) that quickly
reviews any new code (after it is committed) in order to check its compliance
with the library design and coding standard.

It is perfectly acceptable to have an engineer cleaning and committing a contribution written by someone else. But this should be avoided because it is not perennial and it is a waste of resources. Engineers had better focus on tasks beneficial to everyone, including helping during the developments (instead of refactoring after the developments).

9.2 Forge

There is a need for a common repository for: the code itself, its documentation and its related documents (like this one or a web page). The INRIA GForge seems to be the best choice. In addition to a repository, it provides mailing lists and trackers (for the bugs and the feature requests). It may be a good idea to create a mailing list for the developers, and to early use a tracker as a to-do list.

9.3 Revision Control System

Software  A widely used revision control system is Subversion (http://subversion.tigris.org/), which is available on the INRIA GForge.

A decentralized system, like Git (http://git.or.cz/), might be a solution too. The INRIA Gforge team says that it is possible to put a Git repository (through SSH) in a project directory. It is tempting because Git is much more powerful than Subversion (many open source projects are currently migrating from Subversion to Git, or to a similar system like Mercurial).

Repository Management  Another document should expose the management of the repository. Only one aspect of it is reported here. The trunk (or master branch in Git) should always compile and be clean and bug-free (as far as the developers know). Roughly speaking, one could release at any time a version (almost stable) with the trunk. Experimental work should remain in branches—although branches should be avoided with Subversion. Thanks to this rule, one can always trust the trunk and work with it. Any breakage in the trunk would slow down the productivity of all developers (because they most likely use the trunk).

9.4 Compilation

Nowadays, make and the autotools are (fortunately) outdated. The two main replacement tools are CMake (http://www.cmake.org/) and SCons (http://www.scons.org/). With the latter, the SConstruct files (similar to the makefiles) are written in Python. CMake has its own language.
9.5 Unit Testing

Unit testing is strongly advocated, especially for a library. Many tools are available. One is CppUnit, http://cppunit.sourceforge.net/. A good comparison of unit testing frameworks was http://www.gamesfromwithin.com/articles/0412/000061.html, but it is rather old.

9.6 Building the Interface to the Library

A really powerful tool is SWIG (http://www.swig.org/) which can build (almost) automatically a Python (or Ruby, OCaml, Octave, R) interface to a C/C++ code. It essentially supports all C++ features. The interface gives access to all functions, objects and methods of the C++ code.

A complete interface to Python will be built with SWIG. A restricted interface to Matlab will be built with MEX-files.

9.7 Debugging

Valgrind (http://valgrind.org/) is widely used to check that a C/C++ code has no memory leaks and does not access to uninitialized values (tool memcheck—similar to the proprietary software purify). It can also profile the number of instruction performed by any function and any line in a code (tool callgrind). Kcachegrind (http://kcachegrind.sourceforge.net/) may serve as a visualization tool for the Valgrind outputs.

9.8 Miscellaneous Tools

Below is a list of tools that may be useful to the developers.

1. kdiff3: compares and merges two or three files or directories. [http://kdiff3.sourceforge.net/]

2. meld: some people might prefer it over kdiff3. [http://meld.sourceforge.net/, not available under Windows]

3. kdesvn: graphical interface to Subversion. [http://kdesvn.alwins-world.de/, Linux only]

4. format: applied to one or more source files, it indents, removes trailing spaces and checks that no line has more than 78 characters. [∼mallet/bin/format along with ∼mallet/bin/cpp_indent.cpp on Rocquencourt network]

5. ...
9.9 Dependencies

A few libraries (hopefully not too much!) will be needed, at least for linear algebra, random numbers, configuration files and maybe optimization. A detailed comparison of the available solutions should be conducted. Each time, a comparison report should follow to support the final choice.

This page may be helpful: http://www.onumerics.org/oon/

9.9.1 Linear Algebra

A few solutions:
1. CPPLapack: http://sourceforge.net/projects/cpplapack/
2. Flens: http://flens.sourceforge.net/
3. Lapack++: http://sourceforge.net/projects/lapackpp/
4. MTL (matrix template library): http://www.osl.iu.edu/research/mtl/
5. PETSc: http://www-unix.mcs.anl.gov/petsc/petsc-2/ (unfortunately not C++)

These libraries should be compared with respect to their contents, their activity and their documentation. A detailed comparison should be carried out on the basis on what is needed for the library.

9.9.2 Random Numbers

A few solutions:
2. Newran: http://www.robertnz.net/nr03doc.htm

9.9.3 Configuration Files

A few solutions:
2. Talos: http://vivienmallet.net/lib/talos/
9.9.4 Optimization Library

Solutions: ???

Note: LBFGS-B 2.1 available at http://www.ece.northwestern.edu/~nocedal/Software/; version 2.4 exists somewhere on the web

9.10 Library Name

Verdandi is a Norn in Norse mythology whose name means “in the making” or “that which is happening/becoming” (http://en.wikipedia.org/wiki/Verdandi). Verdandi, also representative of the present, comes along with Urd (past) and Skuld (future). The name is relevant because data assimilation somehow reveals the present or what is happening.

Acknowledgement

This document benefits from discussions with Dominique Chapelle and Philippe Moireau. The views on the design are based on experience with the air quality modeling system Polyphemus (http://cerea.enpc.fr/polyphemus/) to which Lin Wu contributed (on the data assimilation part). Claire Mouton also contributed.

Part IV

Appendix

A  A Few Tools Related to Data Assimilation

Below is a list of tools directly related to data assimilation:

1. DART (data assimilation research testbed): http://www.image.ucar.edu/DARES/DART/

2. PALM: http://www.cerfacs.fr/globc/PALM_WEB/

3. Ensemble Kalman Filter (Environment Canada): http://collaboration.cmc.ec.gc.ca/science/arma/enkf/ (Dr. Peter Houtekamer)

B  Excerpts from a Proposal (in French)

Cette section expose les objectifs de la bibliothèque. Elle contient exclusivement des extraits d’une réponse faite à un appel ADT (« action de
développement technologique », programme INRIA) par les équipes-projets CLIME, MACS et MOISE.

B.1 Description

**Titre**  Conception et écriture d’une bibliothèque d’assimilation de données

**Porteurs**  CLIME, MACS, MOISE

**Résumé** Les méthodes d’assimilation de données, qui sont développées et utilisées par plusieurs équipes-projets de l’INRIA, sont suffisamment génériques pour être écrites indépendamment des systèmes auxquels elles s’appliquent. Elles se prêtent donc bien à la constitution d’une bibliothèque dédiée. L’action proposée est une première étape qui consiste à (1) préciser l’architecture logicielle qu’une telle bibliothèque devrait avoir pour satisfaire le plus grand nombre, (2) écrire une première version distribuable. Les motivations sont notamment de faciliter l’application des méthodes d’assimilation à un grand nombre de problèmes, de pérenniser et mutualiser les développements effectués à l’INRIA, et d’améliorer la diffusion des travaux en assimilation dans l’INRIA (entre EPI) et vers l’extérieur.

B.2 Objectifs

**Contexte scientifique** L’assimilation de données consiste à fusionner plusieurs sources d’information sur l’état d’un système pour reconstituer au mieux cet état. En pratique, les sources d’information sont principalement un modèle numérique qui approche l’état du système, et des observations partielles de ce même état. Le système peut être biologique, environnemental, mécanique, …

Les méthodes d’assimilation de données s’écrittent en grande partie indépendamment du système auquel elles sont appliquées, et chaque méthode peut s’appliquer à une grande classe de systèmes. Les méthodes sont donc largement génériques et se prêtent bien à la constitution d’une bibliothèque.

*L’action aura pour objectif de concevoir techniquement et d’écrire une première version de la bibliothèque*. L’étape de conception définira un cadre technique permettant de satisfaire un grand nombre d’utilisateurs, pour une grande classe de problèmes. Ce cadre technique devra en même temps faciliter le développement lié à l’activité de recherche. La première version de la bibliothèque devra être distribuable et suffisamment fournie pour intéresser des utilisateurs. Elle servira aussi de base sûre pour les développements futurs.

**Contenu scientifique** La bibliothèque devra principalement apporter
1. des méthodes d’assimilation pour la prévision et pour la production de réanalyses ;
2. des méthodes inverses (recherche de paramètres) ;
3. des méthodes de prévision d’ensemble.

Afin d’aider au développement de méthodes et à la mise en œuvre de l’assimilation, d’autres composantes seront incluses : opérateurs d’observation classiques, algorithmes de minimisation (par exemple, interfaces avec des implémentations répandues), statistiques d’erreur paramétrées, tests des modèles adjoints et linéaires tangents, quelques modèles et expériences simples pour tester les méthodes, expériences jumelles, génération d’observations synthétiques avec perturbations, méthodes de perturbation de champs, . . .

La bibliothèque visera prioritairement les systèmes de grande dimension.

**Utilité**  La bibliothèque doit servir à

1. faciliter l’application des méthodes à un grand nombre de problèmes ;
2. pérenniser et mutualiser les développements ;
3. améliorer la diffusion des travaux en assimilation dans l’INRIA (entre EPI) et vers l’extérieur.

Il s’agit donc de constituer une bibliothèque accessible à des utilisateurs non initiés à l’assimilation. Les utilisateurs seront en charge de fournir un modèle numérique et des observations avec une interface adéquate.

La bibliothèque sera parallèlement utile à la recherche car elle fournira un cadre de développement rigoureux et pérenne. Ce cadre facilitera aussi les échanges entre EPI. En plus de CLIME, MACS et MOISE, acteurs principaux de l’action, les EPI ASPI (méthodes particulaires), ASCLEPIOS (analyse d’images en biologie), FLUMINANCE (issue de VISTA, analyse d’images spatio-temporelles) et SISYPHE (modélisation, identification, observation et commande, notamment en biologie) sont intéressés.

**Résultat prévu**  Après un an et avec des moyens limités, l’action doit permettre de concevoir techniquement la bibliothèque, en intégrant les contraintes d’un grand nombre de méthodes et d’applications. Une première version de la bibliothèque, distribuable à l’extérieur de l’INRIA et sous licence libre, devra démontrer la validité de la conception.

Si le résultat est convaincant, plusieurs EPI de l’INRIA pourront ensuite contribuer, et une communauté d’utilisateurs devrait se constituer. Le projet devrait alors continuer avec des moyens un peu supérieurs, pour fortement augmenter le contenu de la bibliothèque.
B.3 Description des activités de développement

On rappelle que les objectifs principaux sont :

1. la conception de la bibliothèque, de sorte qu’elle puisse satisfaire un grand nombre d’utilisateurs, qu’elle soit adaptée à une grande classe de problèmes et qu’elle serve de cadre pour les développements issus la recherche ;

2. la réalisation d’une première version de la bibliothèque, distribuable, suffisamment fournie pour présenter un intérêt, et servant de base sûre pour les développements futurs.

Contenu et structure envisagés  Beaucoup de questions techniques sont ouvertes et doivent le rester à ce stade : la conception technique fera partie de l’action et sera arrêtée après considération de toutes les contraintes et des solutions adéquates. On expose ici brièvement le contenu de la bibliothèque et des mises en œuvre techniques possibles (déjà expérimentées dans le système Polyphemus).

La bibliothèque sera composée de classes et/ou de fonctions ayant des interfaces et des implémentations cohérentes. Il ne s’agit donc pas de rassembler des codes existants.

La bibliothèque contiendra en premier lieu des méthodes d’assimilation et d’estimation de paramètres, c’est-à-dire probablement des classes qui produiraient des analyses, des prévisions et/ou des paramètres inversés, étant donné un modèle, des observations, un opérateur d’observation et les statistiques d’erreur. Ces derniers seraient fournis par l’utilisateur sous forme de classes.

Des éléments génériques devront s’ajouter pour aider au développement : opérateurs d’observation classiques (par exemple, interpolation d’un champ 3D), algorithmes de minimisation (par exemple, interfaces avec des implémentations répandues), statistiques d’erreur paramétrées, tests des modèles adjoints et linéaires tangents, quelques modèles et expériences simples pour tester les méthodes, expériences jumelles, génération d’observations synthétiques avec perturbations, méthodes de perturbation de champs.

Les langages de programmation ne sont pas encore déterminés. On peut penser qu’un langage orienté objet sera nécessaire à la manipulation des modèles et des observations. Un langage compilé (par exemple, le C++) et un langage interprété (par exemple, Matlab, Octave ou Python) pourraient former un couple idéal pour répondre aux besoins de tous les utilisateurs.

References


