YAO:
a Software for Variational Data Assimilation using Numerical Models

Luigi Nardi[1,2], Charles Sorror[1], Fouad Badran[1,2], and Sylvie Thiria[1]

[1] LOCEAN, Laboratoire d'Océanographie et du Climat:
Expérimentations et Approches Numériques.
Unité Mixte de Recherche 7159 CNRS / IRD /
Université Pierre et Marie Curie/MNHN.
Institut Pierre Simon Laplace.
4, place Jussieu 75252 Paris Cedex 05, France

[2] CNAM-CEDRIC, Centre d'Etude et De Recherche en Informatique du CNAM
292, rue St Martin 75141 Paris Cedex 03, France

{luigi.nardi,charles.sorror,fouad.badran,sylvie.thiria}
@locean-ipsl.upmc.fr
http://www.locean-ipsl.upmc.fr/

Abstract. Variational data assimilation consists in estimating control parameters of a numerical model in order to minimize the misfit between the forecast values and some actual observations. The gradient based minimization methods require the multiplication of the transpose jacobian matrix (adjoint model), which is of huge dimension, with the derivative vector of the cost function at the observation points. We present a method based on a *modular graph* concept and two algorithms to avoid these expensive multiplications. The first step of the method is a propagation algorithm on the graph that allows computing the output of the numerical model and its linear tangent, the second is a backpropagation on the graph that allows the computation of the adjoint model. The YAO software implements these two steps using appropriate algorithms. We present a brief description of YAO functionalities.

Key words: Data assimilation, numerical model, modular graph, adjoint model, automatic differentiation, backpropagation

1 Introduction

Numerical models are widely used as a tool for studying physical phenomena. A direct numerical model is a discretization of the equations that represent the physical phenomenon under study. The space structure of the phenomenon can be 1D, 2D, 3D; often an additional dimension is added to represent the time evolution. Most of the time the model is used to forecast or analyse the evolution of the phenomenon. Since the model is imperfect, discrepancy between its forecast

values and actual observations may be important due to model parametrization, numerical discretization, uncertainty on the initial conditions and boundary conditions. New methods which use both the direct model of the phenomenon and inverse problem method have been introduced to overcome this difficulty, the so-called *data assimilation*. Data assimilation seeks a good compromise between the actual observations at some points of the space/time location and the corresponding outputs of the numerical model. The observations constrain the control parameters (initial conditions, model parameters, ...) in order to force the direct model to reproduce the desired behavior.

One can distinguish between two types of data assimilation methods: *sequential* and *variational* ([1–6]). The present paper is dedicated to variational methods which are more suited for observations which are not given regulary in time and space. Variational methods consist in defining a cost function J which measures the misfit between the direct numerical model outputs and the observations. Their aim is to minimize the function J which depends on the control parameters. This can be done by operating a local minimization using a gradient method. Normally, the user programs the direct (dynamic) model, computes the gradient of the cost function by programming the adjoint model and schedules the operations of minimization according to the selected scenario. From the data-processing point of view this yields two types of problems:

- If a direct model program has been implemented, it is necessary, for the assimilation purpose, to implement the program which provides the adjoint model and sometimes also its tangent linear model.
- Once all these models have been implemented, it is necessary, to schedule the various calculations according to a certain scenario and to the chosen minimization method.

The first problem leads to use automatic differentiation softwares ([7–9]) and the second problem to design specific softwares ([10]). The YAO software, which we develop at LOCEAN laboratory, concerns these two tasks and aims to deal with the two previous problems simultaneously. With YAO, the user specifies, using specific directives, the type of discretization and the specification of the direct model. YAO generates the direct model, the tangent linear model and the adjoint model. It also allows to choose, according to the specific scenario, an implementation for the minimization function J.

In the following, section 2 deals with the notations of variational data assimilation; section 3 introduces the *modular graph* which is the YAO basic formalism; section 4 presents the basics algorithms used by YAO; section 5 deals with the decomposition of an application by a *modular graph*; section 6 makes a brief presentation of YAO functionalities and some applications which were already implemented; section 7 presents a simple example showing how to get a YAO graph from a direct model.

2 Theoretical Principles and Notations

Variational assimilation requires the knowledge of a numerical model, the so-called direct model M which describes the physical phenomenon evolution under study. If we take for example a geophysical problem, the direct model allows to link together the geophysics variables and the observations. The assimilation consists in modifying the control parameters so that the model reproduces the observations as good as possible. The control parameters can be for example the initial conditions or unknown parameters of M. In this section, we present the formal mathematical notations. We adopt the formalism and the notations presented in [11]:

- M: direct model describing the evolution (that is in general nonlinear) between two time steps of discretization t_i, t_{i+1}.
- $\boldsymbol{x}(t_0)$: initial input state vector - we suppose thereafter that it corresponds to the control variables.
- $M_i(\boldsymbol{x}(t_0))$ or $M(t_0, t_i)$: state model at time t_i beginning from t_0. We will denote $\boldsymbol{x}(t_i) = M_i(\boldsymbol{x}(t_0))$.
- $\boldsymbol{M}(t_i, t_{i+1})$: tangent linear matrix which is the jacobian of model M calculated at $\boldsymbol{x}(t_i)$.
- The tangent linear matrix of the model M_i calculated at $\boldsymbol{x}(t_0)$ is defined by:

$$\boldsymbol{M}_i(\boldsymbol{x}(t_0)) = \prod_{j=i-1}^{0} \boldsymbol{M}(t_j, t_{j+1}) \ .$$

- The adjoint matrix of the model M_i calculated at $\boldsymbol{x}(t_0)$ is defined by:

$$\boldsymbol{M}_i(\boldsymbol{x}(t_0))^T = \prod_{j=0}^{i-1} \boldsymbol{M}(t_j, t_{j+1})^T \ .$$

- $\boldsymbol{x}^b$: a background vector which is an estimate of the vector $\boldsymbol{x}(t_0)$.
- $\boldsymbol{y}^o$: set of observations at different time. The vector $\boldsymbol{y}_i^o$ thus represents the observations at time t_i, this vector can be empty if there are no observations at time t_i.

The model M allows to estimate quantities which are generally observed with the *observation operator* (H). In the field of geophysics this operator allows, for example, to compare the model M outputs which calculates the temperature at sea surface with observations recorded by a satellite radiometer. We denote:

- H: *observation operator* which allows to calculate, starting from the direct model outputs at $\boldsymbol{x}(t_i)$, $\boldsymbol{y}_i$=$H(\boldsymbol{x}(t_i))$, the quantity $\boldsymbol{y}_i$ being the equivalent of the observation variables $\boldsymbol{y}_i^o$. It is supposed thereafter that: $\boldsymbol{y}_i^o$=$\boldsymbol{y}_i$+ε_i (ε_i is a random variable of null average).
- $\boldsymbol{H}_i$: tangent linear model matrix of the H operator calculated at $\boldsymbol{x}(t_i)$.

The assimilation consists in minimizing a cost function J which measures the misfit between the direct numerical model outputs and the observations by improving the control variables. Generally the cost function is defined as:

$$J(\boldsymbol{x}(t_0)) = \frac{1}{2}\sum_{i=1}^{n}(\boldsymbol{y}_i - \boldsymbol{y}_i^o)^T \boldsymbol{R}^{-1}(\boldsymbol{y}_i - \boldsymbol{y}_i^o) + \frac{1}{2}(\boldsymbol{x}(t_0) - \boldsymbol{x}^b)^T \boldsymbol{B}^{-1}(\boldsymbol{x}(t_0) - \boldsymbol{x}^b) \ . \quad (1)$$

With $\boldsymbol{y}_i = H(\boldsymbol{x}(t_i))$, $\boldsymbol{R}$ the covariance matrix estimation on the observation errors ε_i, $\boldsymbol{B}$ the covariance matrix estimation on the error background vector $\boldsymbol{x}^b$ and n the total number of time intervals. The gradient $\nabla_{\boldsymbol{x}_0} J$ is equal to:

$$\nabla_{\boldsymbol{x}(t_0)} J = \sum_{i=1}^{n} \boldsymbol{M}_i^T(\boldsymbol{x}(t_0)) \boldsymbol{H}_i^T [\boldsymbol{R}^{-1}(\boldsymbol{y}_i - \boldsymbol{y}_i^o)] + \boldsymbol{B}^{-1}(\boldsymbol{x}(t_0) - \boldsymbol{x}^b) \ . \quad (2)$$

The minimization procedure, which is a gradient type method, is carried out by choosing a particular implementation among the set of those proposed by the optimization techniques([12]). These methods need to calculate the cost function gradient (2) with respect to the control parameters; Fig. 1 depicts the basic iteration.

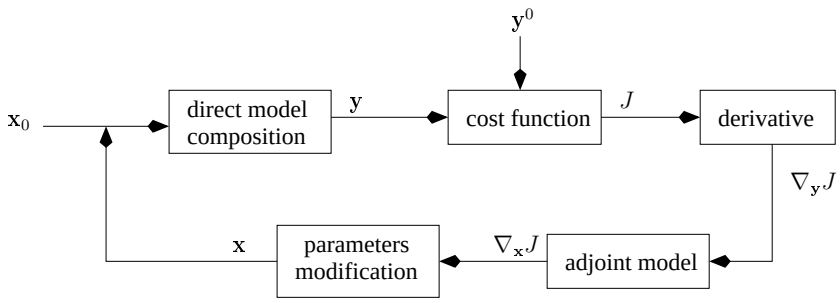

Fig. 1. Basic iteration of the variational data assimilation: the misfit between the direct model output $\boldsymbol{y}$ and the observations $\boldsymbol{y}^o$ is defined by the cost function J. The derivative vector $\nabla_{\boldsymbol{y}} J$ is used to compute the matrix product defined by the first term of expression (2), $\boldsymbol{x}_0$ is the vector $\boldsymbol{x}(t_0)$ at the first iteration.

In order to facilitate the convergence in some specific problems, we can use an approximate gradient descent method, called *incremental algorithm* [13]. The *incremental algorithm* consists in modifying locally the J function. The minimization of the function J entails to initialize the minimization algorithm with an initial state $\boldsymbol{x}^g(t_0)$, so that the control vector parameter we look for can be defined by $\boldsymbol{x}(t_0) = \boldsymbol{x}^g(t_0) + \boldsymbol{\delta x}(t_0)$. We use the tangent linear approximation:

$$\boldsymbol{y}_i = H(M_i(\boldsymbol{x}(t_0))) \simeq H(M_i(\boldsymbol{x}^g(t_0))) + \boldsymbol{H}_i(\boldsymbol{M}_i(\boldsymbol{x}^g(t_0)))\boldsymbol{\delta x}(t_0) \ . \quad (3)$$

If we pose $\boldsymbol{d}_i = \boldsymbol{y}_i^o - H(M_i(\boldsymbol{x}^g(t_0)))$ we can express J as:

$$J[\boldsymbol{\delta x}(t_0)] = \frac{1}{2}\sum_{i=1}^{n}[\boldsymbol{H}_i\boldsymbol{M}_i(\boldsymbol{x}^g(t_0))\boldsymbol{\delta x}(t_0) - \boldsymbol{d}_i]^T\boldsymbol{R}^{-1}[\boldsymbol{H}_i\boldsymbol{M}_i(\boldsymbol{x}^g(t_0))\boldsymbol{\delta x}(t_0) - \boldsymbol{d}_i] + \\ + \frac{1}{2}[\boldsymbol{\delta x}(t_0) - (\boldsymbol{x}^b - \boldsymbol{x}^g(t_0))]\boldsymbol{B}^{-1}[\boldsymbol{\delta x}(t_0) - (\boldsymbol{x}^b - \boldsymbol{x}^g(t_0))]\ . \tag{4}$$

This new formulation of J represents a quadratic expression with respect to $\boldsymbol{\delta x}(t_0)$ and corresponds to a $J(\boldsymbol{\delta x}(t_0))$ approximation around $\boldsymbol{x}^g(t_0)$. The problem is to minimize $J(\boldsymbol{\delta x}(t_0))$ depending on the vector $\boldsymbol{\delta x}(t_0)$ only, since the other function terms of (4) are constants. This phase of minimization is denoted as the *internal loop*. In each internal loop iteration, we have to calculate the gradient:

$$\nabla_{\boldsymbol{\delta x}(t_0)}J = \sum_{i=1}^{n}\boldsymbol{M}_i^T(\boldsymbol{x}^g(t_0))\boldsymbol{H}_i^T\boldsymbol{R}^{-1}\left[\boldsymbol{H}_i\boldsymbol{M}_i(\boldsymbol{x}^g(t_0))\boldsymbol{\delta x}(t_0) - \boldsymbol{d}_i\right] + \\ \boldsymbol{B^{-1}}\left[\boldsymbol{\delta x}(t_0) - (\boldsymbol{x}^b - \boldsymbol{x}^g(t_0))\right]\ . \tag{5}$$

After processing an internal loop (minimization iterations) for computing $\boldsymbol{\delta x}(t_0)$, we restart the minimization algorithm at the new initial state $\boldsymbol{x}^g(t_0)) + \boldsymbol{\delta x}(t_0)$ and run again the internal loop for some more iterations. This initial state setting phase is called *external loop*.

In this paper we present the incremental method which uses a two-stage scenario for the minimization process. Others scenarios, as the *Dual* formalism which takes into account the possible error of model M, lead to others and sometime more sophisticated computations([14, 15]). The goal of this paper is not to present all the possible scenarios but to focus on the complexity of the algorithm needed for their implementation. Equations (2) and (5) show that gradient calculations require matrix products of type $\boldsymbol{M}_i^T\boldsymbol{\delta y}$ and $\boldsymbol{M}_i\boldsymbol{\delta x}$ for any i. However, generally the matrices $\boldsymbol{M}_i^T$ and $\boldsymbol{M}_i$ being of huge dimensions, it is not so easy to calculate their product. We present in section 4 two algorithms to compute these matrix products without the explicit knowledge of the matrices.

3 Modular Graph

We define the following terms:

- *Module*: a module F is an entity of computation; it receives an input vector and calculates an output vector. A module receives inputs from other modules or from the external graph context and it transmits outputs to other modules or to the external graph context.
- *Basic connection*: if the jth output of a module F_p is transmitted to the ith input of a module F_q, we can modelize this transmission by a connection between the output j of the module F_p and the input i of the module F_q

which we denote thereafter by (i, q) and (j, p). We call this connection *basic connection*. Data transmission towards the external context of the graph will be represented by a *basic connection* starting from a module output and ending at a special node called *output data point*. Data transmission towards the interior of the graph will be represented by a *basic connection* starting from a special node called *input data point* and ending at an input of a module.

A *modular graph* is a set of several interconnected modules. The modules represent the graph vertices; an arc from module F_p to the module F_q means that there exists at least one *basic connection* from F_p to F_q (Fig. 2a). The modular graph thus describes the scheduling of the modules execution (Fig. 2b).

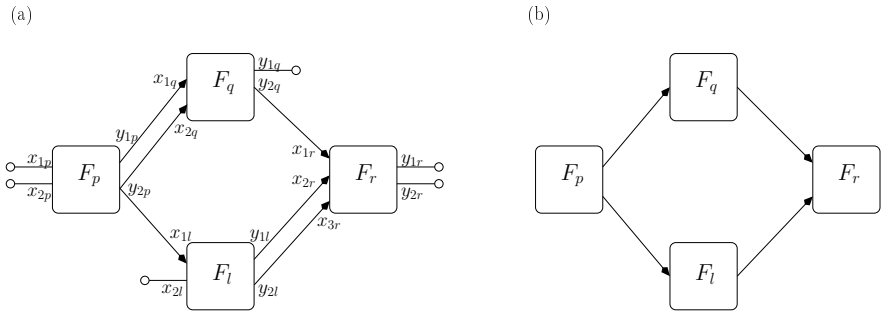

Fig. 2. (a) Basic connections between modules, *input data points* $\boldsymbol{x}^T = (x_{1p}, x_{2p}, x_{2l})$ and *output data points* $\boldsymbol{y}^T = (y_{1q}, y_{1r}, y_{2r})$. (b) Corresponding modular graph.

The modular graph summarizes the sequential order of the computations: an arc from F_p to F_q indicates that F_q must start its execution only after F_p has been executed. The modular graph is an acyclic graph, so it contains three types of vertex:

- The modules with no predecessors in the graph, receive data from *input data points* only.
- The modules with no successors in the graph, transmit outputs only to *output data points*.
- The internal graph modules necessarily receive inputs from one or several other modules and eventually from *input data points* and transmit results to their modules' successors or *output data points*.

The input data set of a module F_p constitutes a vector denoted $\boldsymbol{x}_p$ and its output data set constitutes a vector denoted $\boldsymbol{y}_p$ ($\boldsymbol{y}_p = F_p(\boldsymbol{x}_p)$). As a consequence, a module F_q can be executed only if its input vector $\mathbf{x_q}$ has already been processed, which implies that all its predecessor modules F_p have been executed beforehand. Since the modular graph is acyclic, it is then possible to find a module's order (the topological order) which respects the following property: if $F_p \rightarrow F_q$ is an

arc of the modular graph then F_p precedes F_q in the order of computation. The topological order allows to correctly propagate the calculation through the graph from the input points to the output points, all the graph *input data points* being initialized by the external context. The propagation of intermediate computations following the topological order is called *forward* procedure. This procedure gives the way to produce the correct final value of the global computation of the direct model. If we denote by $\boldsymbol{x}$ the vector corresponding to all the graph *input data point* values, the *forward* procedure allows to calculate the vector $\boldsymbol{y}$ corresponding to all graph *output data point* values. The modular graph defines a global function Γ and makes it possible to compute $\boldsymbol{y} = \Gamma(\boldsymbol{x})$.

4 Computation of the Tangent Linear and the Adjoint of the Global Function $\boldsymbol{\Gamma}$

4.1 Tangent Linear Computation

We now denote by Γ a graph (composed by its F_q modules) and assume that each F_q can compute the matrix product $\boldsymbol{dy}_q = \boldsymbol{F}_q \boldsymbol{dx}_q$, where $\boldsymbol{dx}_q$ is the perturbation of $\boldsymbol{x}_q$ and $\boldsymbol{F}_q$ is the jacobian matrix of F_q calculated at $\boldsymbol{x}_q$. It is then possible to compute, as for the *forward* procedure, the product $\boldsymbol{dy} = \boldsymbol{\Gamma} \boldsymbol{dx}$ where $\boldsymbol{dx}$ represents the perturbation associated with all the input vectors x_q and $\boldsymbol{\Gamma}$ the jacobian matrix of Γ computed at an input vector $\boldsymbol{x}$. This computation is done on the modular graph using the following algorithm:

***Lin_forward* algorithm.** Before determining the linear tangent of the global function Γ for a given input data $\boldsymbol{x}$, all the inputs of the different modules x_{ip} have to be determined. This is done by running the forward procedure with $\boldsymbol{x}$ as input.

1. Initializing by the external context, the perturbation $\boldsymbol{dx}$, by assigning to each *input data points* i of the graph its corresponding perturbation $\boldsymbol{dx}_i$.
2. Passing through all the modules by following the topological order. For each module F_q we consider its input perturbation vector $\boldsymbol{dx}_q$. This can be done by transmitting computed perturbations from the output of its predecessors modules or from those initiated by the external context. Then we compute $\boldsymbol{dy}_q = \boldsymbol{F}_q \boldsymbol{dx}_q$ (the jacobian $\boldsymbol{F}_q$ is computed at the point $\boldsymbol{x}_q$).
3. Recovering the vector result $\boldsymbol{dy}$ on the graph *output data points*. This vector represents the perturbation of the global function Γ.

4.2 Adjoint Computation

As in the case of the tangent linear model, we suppose that for each module F_p, with an input vector $\boldsymbol{x}_p$ and receiving in its output points a perturbation vector $\boldsymbol{dy}_p$, we can calculate the matrix product $\boldsymbol{dx}_p = \boldsymbol{F}_p^T \boldsymbol{dy}_p$. $\boldsymbol{F}_p^T$ is the transposed jacobian matrix of the module F_p calculated at the point $\boldsymbol{x}_p$. We

can prove that it is possible to compute the matrix product $\boldsymbol{dx} = \boldsymbol{\Gamma}^T \boldsymbol{dy}$, where $\boldsymbol{\Gamma}^T$ is the transposed jacobian of the global model Γ calculated at the input vector $\boldsymbol{x}$ and $\boldsymbol{dy}$ is a perturbation vector defined at each *output data point*. This computation is done by passing through the modules of the graph in the reverse topological order (*backpropagation*). The *backward* algorithm, requires, for each module F_p, the definition of two vectors $\boldsymbol{\alpha}_p$ and $\boldsymbol{\beta}_p$. The vector $\boldsymbol{\alpha}_p$ has the same dimension as the module F_p input vector and we denote by α_{ip} its ith input element. The vector $\boldsymbol{\beta}_p$ has the same dimension as the module F_p output vector and we denote by β_{jp} its jth output element. Moreover, the α parameters are also defined for the *output data points* and the β parameters are defined for the *input data point*. We denote by α_i the parameter of the *output data points* i and by β_j the parameter of the *input data point* j.
If (j, p) is the index of the jth output of F_p, we denote by:

- $SUCCM(j, p)$ the set of indices (i, q), ith input of module F_q, which take the value of (j, p) as input.
- $SUCCO(j, p)$ the set of all the *output data points* which take the value of (j, p) as input.

If j is an *input data point* we denote by $SUCCI(j)$ the set of indices (i, q), ith input variables of the modules F_q, which take their value from the j *input data point*.

***Backward* algorithm.** Before running this algorithm, all inputs $\boldsymbol{x}_{ip}$ of all modules F_p should have been calculated. For that, it is necessary to run the *forward* algorithm with the input vector $\boldsymbol{x}$. For every *output data point* j we suppose that its corresponding perturbation is already defined by $\boldsymbol{dy}_j$.

1. Initializing the parameters α_i relative to the graph *output data points* i by assigning $\alpha_i = \boldsymbol{dy}_i$.
2. Passing through all the modules in the reverse topological order. For each module F_p computes $\boldsymbol{\beta}_p$ and $\boldsymbol{\alpha}_p$ as follow:
   - For all its output indices (j, p), performs the following operations (in order to compute $\boldsymbol{\beta}_p$):
     - Assign $\beta_{jp} = 0$.
     - If $SUCCM(j, p)$ is not empty then compute $\beta_{jp} = \sum_{(i,q) \in SUCCM(j,p)} \alpha_{iq}$.
     - If $SUCCO(j, p)$ is not empty then compute $\beta_{jp} = \beta_{jp} + \sum_{i \in SUCCO(j,p)} \alpha_i$.
   - Computes $\boldsymbol{\alpha}_p = \boldsymbol{F}_p^T \boldsymbol{\beta}_p$, where $\boldsymbol{F}_p^T$ is the transpose of the jacobian matrix computed at point $\boldsymbol{x}_p$.
3. For each j *input data point*, compute $\beta_j = \sum_{(i,q) \in SUCCI(j)} \alpha_{iq}$.

The vector $\boldsymbol{dx}$, whose components are the β_j of all the graph *input data points*, verifies $\boldsymbol{dx} = \boldsymbol{\Gamma}^T \boldsymbol{dy}$. $\boldsymbol{dy}$ is the vector whose components are the values $\boldsymbol{dy}_i$ defined at the graph *output data points*.

Remark 1. The two algorithms *lin_forward* and *backward* first suppose that we can compute the tangent linear and the adjoint of each module F_p. Modules could have very different complexities. In a simple case, where the module is an analytical function, we can calculate the jacobian matrix $\boldsymbol{F}_p$ explicitly and calculate the product $\boldsymbol{F}_p \boldsymbol{dx}_p$ and $\boldsymbol{F}_p^T \boldsymbol{dy}_p$. Moreover it is important to define the modular graph in order to have "small" modules (small entity of computation) so the analytical calculation of $\boldsymbol{F}_p$ becomes easy. Concerning more complex modules, we can use programs which make these computations (i.e. a code obtained using automatic differentiation software [7–9], or even another modular graph). So the modular graph formalism, and its related algorithms, makes possible to merge different numerical codes to build complex numerical models.

5 Representation of an Application with a Modular Graph

Running simulations or data assimilation using an operational numerical model $M_i(x(t_0)) = M(t_{i-1}, t_i) \circ M(t_{i-2}, t_{i-1}) \circ \ldots \circ M(t_0, t_1)$ require the definition of a modular graph representing the sequence of the computations. A numerical model operates on a discreet grid, where the physical process is computed at each time step and at each grid point. As the phenomenon under study is quite the same at each grid point, there is a large amount of repetitivity. So the modular graph Γ, associated with the numerical model M, must take into account this concept of repetitivity:

- A modular sub-graph (Γ_g) describes all the computations needed at time t for a given grid point (Fig. 3a).
- The (1D, 2D, 3D) graph is thus a modular graph whose vertices are the sub-graph Γ_g and the arcs represent the information exchanges between them (Fig. 3b).
- The complete graph Γ, which takes into account a time interval, is obtained by duplicating the graph as long as necessary.

As defined previously, we used the same concept of *input data* and *output data points* for each module of the complete graph. The basic connections coming from the external context of Γ could be, for example, initializations or boundary conditions. Outgoing basic connections transmit their values to compute, for example, a cost function.

6 YAO Presentation

We have presented in the above sections the basic concepts and algorithms of the YAO software. We present, briefly in this section, the overall structure of YAO and the various directive files that allow to generate applications. YAO permits the user to work with 1D, 2D, 3D problems and the time dimension. The user focuses on the application design and does not care about hard programming work. He specifies its problem using a specific language (YAO description

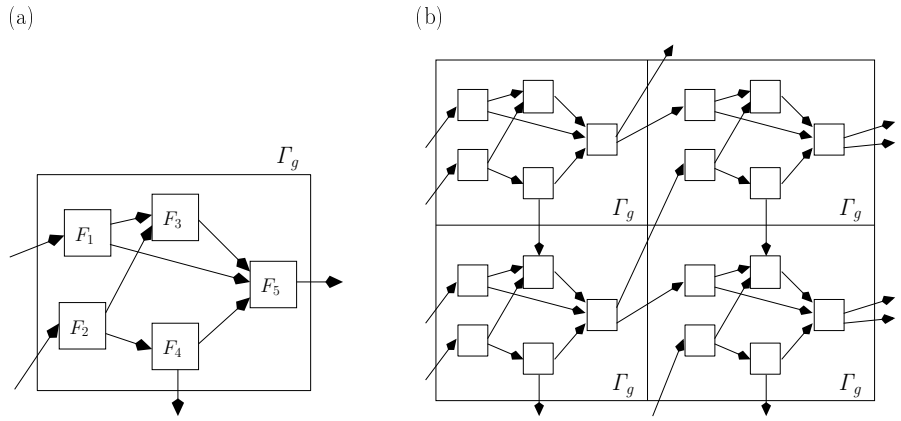

Fig. 3. Two graph abstraction levels: at the lower level (Fig. a), we build the graph Γ_g; at the space level (Fig. b), the same graph Γ_g is repeated for each grid point (2D in this example). The space connections between the Γ_g graphs correspond to the basic connections between the modules.

language) and YAO automatically generates the associated modular graph. As already indicated, the description of the modular graph and the three related algorithms, is similar to a program giving the direct numerical model, the tangent linear and the adjoint. Moreover, YAO allows to define specific minimization scenarios. Figure 4 gives a schematic representation of the basic YAO architecture. The description and the instructions files have to be written by the user of

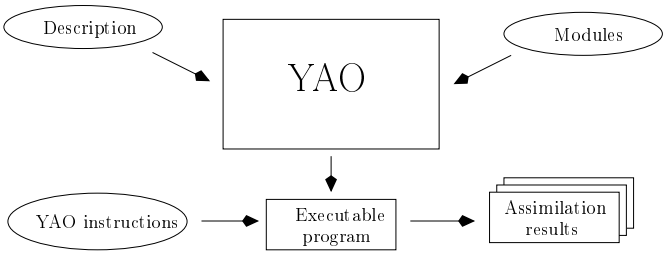

Fig. 4. Schematic representation of YAO: code generation starting from the description file (which is a model specification) and the modules files. This generated code (executable program) runs with an instructions file which will control the results production.

YAO as well as the module specifications and the jacobian of each module. YAO takes into account these files and generates an executable program. We briefly present the various YAO aspects.

The Description File. This file contains the YAO directives which define the direct numerical model characteristics. In particular, it is necessary:

- To define the numerical model time steps (denoted as the *trajectory*).
- To define the discreet space grid mesh (denoted *spaces*) and its dimension (1D, 2D, 3D).
- To introduce all the information related to the cost function: observations, covariance matrix, ...
- To define the modules, specifying for each one the number of inputs/outputs, its participation in the cost function, ...
- To build the graph by defining the basic connections between the modules.
- To indicate the computational order (the order in which the grid points have to be considered).

The description file contains all these information which are used by YAO to generate the executable code.

The Instructions File. This file contains specific instructions (YAO instruction language) for running the model in a dedicated configuration (duration of the simulation, time increments, physical size of the space, initial values of the parameters, ...). The YAO instruction language allows to control the execution flow, to modify some parameters during the runtime and to introduce a background for the cost function.

The modules Files. These files contain the source code in which the physical laws of the numerical model, the input parameters and the jacobian are programmed. Figure 4 displays the YAO architecture: the part into the large rectangular frame contains the YAO procedures for generating the modular graph. YAO also uses the description file and the module files presented above to generate the executable program (which are out of the frame in Fig. 4). Once the executable program is created the instructions file is used to execute the user instructions. Modifying a model consists in changing some modules or some YAO directives in the description files. Modifying an execution of a YAO application consists in changing the YAO instructions in the instructions file.

Although YAO works in C/C++, it is nevertheless possible to make links with other languages. Since that YAO does not care about the size of each module, the user may choose the model's decomposition. From a practical point of view, this may depend on the module decomposition, and also on how the gradient of a module is computed. As already indicated, it is possible, in addition to YAO, to use an automatic differentiator rather than to code manually the Jacobian matrix. YAO provides some functionalities of an integrated tool. For example, it manages interfacing with a minimizer such as M1QN3 ([16]); it can deal with multilayer perceptrons; it includes a general cost function taking into account background and covariance operators. Moreover, YAO manages multi-trajectories and multi-dimensional (up to 3D) computations. YAO has already been tested with success on several models in oceanography. It was applied in the following applications:

– **Ocean color**: variational inversion of multi-spectral satellite ocean color measurements for the restitution of the chlorophyll-a [17, 18].
– **Marine acoustics**: variational inversion of sound speed profile and retrieval of geoacoustic parameters (celerity, density, attenuation, ...) [19, 18].
– **PISCES**: ocean color variational data assimilation in a biogeochemical model [20].

7 Numerical Example: The Shallow-Water

This section presents on a simple example the necessary steps to get the YAO graph of the direct model. At the end of the specification, YAO can generate the direct code in C++, the linear tangent code and the adjoint code and is able to make some assimilation experiments. We chose the two dimensions (2D) shallow-water model in the horizontal plane (x,y), also called Saint-Venant model, which arises from the vertical integration of three dimensions (3D) Navier-Stokes equations. This model describes the linear flow of a nonviscous fluid in shallow water environment with a free surface. In the present study, focus is given on the internal mode of the two lauer fluid whose densities are slightly different. The evolution is described by the following system of partial differential equations:

$$\begin{aligned}
\frac{\partial u}{\partial t} &= -g^* \cdot \frac{\partial h}{\partial x} + f \cdot v - \gamma \cdot u \\
\frac{\partial v}{\partial t} &= -g^* \cdot \frac{\partial h}{\partial y} - f \cdot u - \gamma \cdot v \\
\frac{\partial h}{\partial t} &= -H \cdot \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right)
\end{aligned}$$

– u and v are the horizontal velocities on axes x,y.
– h is the amplitude (the height) of the free surface.
– g^* is the reduced gravity.
– f is the Coriolis parameter.
– γ is a dissipation coefficient.
– H is the average height of the water.

The system of partial differential equations is resolved spatially by using the Arakawa C grid. For the temporal axis we use a leap frog discretization followed by an Asselin filter to ensure stability. The spatial discretization is based on a regular 2D grid. After initialization the direct numerical model is the following:

– Dynamic variables:

$$u_{ijt} = \hat{u}_{ijt-2} + 2\Delta t \left(\frac{-g^*}{\Delta x} \left[h_{i+1jt-1} - h_{ijt-1} \right] + \right. \tag{6}$$

$$\left. \frac{f}{4} \left[v_{ijt-1} + v_{ij+1t-1} + v_{i+1jt-1} + v_{i+1j+1t-1} \right] - \gamma \cdot \hat{u}_{ijt-2} \right)$$

$$v_{ijt} = \hat{v}_{ijt-2} + 2\Delta t \left(\frac{-g^*}{\Delta y} \left[h_{ijt-1} - h_{ij-1t-1} \right] - \right. \tag{7}$$

$$\left. \frac{f}{4} \left[u_{i-1j-1t-1} + u_{i-1jt-1} + u_{ij-1t-1} + u_{ijt-1} \right] - \gamma \cdot \hat{v}_{ijt-2} \right)$$

$$h_{ijt} = \hat{h}_{ijt-2} - 2\Delta t \cdot H \left(\frac{u_{ijt-1} - u_{i-1jt-1}}{\Delta x} + \frac{v_{ij+1t-1} - v_{ijt-1}}{\Delta y} \right). \tag{8}$$

– Asselin filter:

$$\hat{u}_{ijt} = u_{ijt} + \alpha(\hat{u}_{ijt-1} - 2u_{ijt} + u_{ijt+1}) \tag{9}$$

$$\hat{v}_{ijt} = v_{ijt} + \alpha(\hat{v}_{ijt-1} - 2v_{ijt} + v_{ijt+1}) \tag{10}$$

$$\hat{h}_{ijt} = h_{ijt} + \alpha(\hat{h}_{ijt-1} - 2h_{ijt} + h_{ijt+1}) \tag{11}$$

where $\hat{h}$, $\hat{u}$ and $\hat{v}$ are intermediate variables. In the following we present the directives of the YAO description file, which define the *basic connections* related to (6–11) and allow to generate the shallow-water modular graph. We assume that the model is defined on a 50×50 grid (Δx=$\Delta y = 5000$ meters) and on a 100 time step trajectory ($\Delta t = 1500$ seconds, about 1 day and 17 hours).

```
traj shallow_trajectory 1 100
space shallow_space 50 50 shallow_trajectory

modul Hfil space shallow_space input 3 output 1 tempo cout target
modul Ufil space shallow_space input 3 output 1 tempo
modul Vfil space shallow_space input 3 output 1 tempo
modul Hphy space shallow_space input 5 output 1 tempo
modul Uphy space shallow_space input 7 output 1 tempo
modul Vphy space shallow_space input 7 output 1 tempo

ctin Hfil 1 from Hfil 1  i   j   t-1
ctin Hfil 2 from Hphy 1  i   j   t-1
ctin Hfil 3 from Hphy 1  i   j   t
ctin Ufil 1 from Ufil 1  i   j   t-1
ctin Ufil 2 from Uphy 1  i   j   t-1
ctin Ufil 3 from Uphy 1  i   j   t
ctin Vfil 1 from Vfil 1  i   j   t-1
ctin Vfil 2 from Vphy 1  i   j   t-1
ctin Vfil 3 from Vphy 1  i   j   t
ctin Hphy 1 from Hfil 1  i   j   t-1
ctin Hphy 2 from Uphy 1  i   j   t-1
ctin Hphy 3 from Uphy 1  i-1 j   t-1
ctin Hphy 4 from Vphy 1  i   j   t-1
ctin Hphy 5 from Vphy 1  i   j+1 t-1
ctin Uphy 1 from Ufil 1  i   j   t-1
ctin Uphy 2 from Hphy 1  i   j   t-1
ctin Uphy 3 from Hphy 1  i+1 j   t-1
ctin Uphy 4 from Vphy 1  i   j   t-1
ctin Uphy 5 from Vphy 1  i   j+1 t-1
```

```
ctin Uphy 6 from Vphy 1  i+1 j   t-1
ctin Uphy 7 from Vphy 1  i+1 j+1 t-1
ctin Vphy 1 from Vfil 1  i   j   t-1
ctin Vphy 2 from Hphy 1  i   j-1 t-1
ctin Vphy 3 from Hphy 1  i   j   t-1
ctin Vphy 4 from Uphy 1  i-1 j-1 t-1
ctin Vphy 5 from Uphy 1  i-1 j   t-1
ctin Vphy 6 from Uphy 1  i   j-1 t-1
ctin Vphy 7 from Uphy 1  i   j   t-1

order shallow_space
  order YA1 YA2
        Hphy Uphy Vphy Hfil Ufil Vfil
  forder
forder
```

The directive **traj** defines the trajectory called *shallow_trajectory* composed by an initialization phase (1 time step) and a set of 100 time steps during which the model run.
The directive **space** defines a space called *shallow_space*. The space is defined by two axes of dimension 50×50 and it is linked to the trajectory *shallow_trajectory*. YAO references the axes by *YA1* for the first and *YA2* for the second. These references will be used thereafter to indicate the order to traverse the space in the directive **order**.
The directive **modul** allows to declare modules. The YAO grammar defines keywords to figure out some attributes that characterize the module. The keyword **modul** is followed by the name of the module (for instance *Hfil*) and then the keyword **space** followed by the name of the space linked with the module. The *input* and *output* attributes allow to specify the number of input and output of the module. *tempo* indicates that YAO must store the computed states on all the time steps of the trajectory; this could be useful for derivative computation and for referencing previous time steps. The keyword *target* allows to control the outputs of this module, therefore it is the target of our assimilation process. The term *cost* means that the output of this module is related to some observations and it will take part of the cost function computation. The implementation of the modules is done in the module files as shown in Fig. 4. In this example the modules (*Hphy*, *Uphy*, *Vphy*, *Hfil*, *Ufil*, *Vfil*) represents (h,u,v,$\hat{h}$, $\hat{u}$, $\hat{v}$).
The directive **ctin** is used to create the *basic connections* of the graph. The input of the first module, at a current point (i,j,t) of the discretized space, takes its value from the output of the second module at a specified point (refer to after the module name). The numbers after the first module indicates the number of its specific input, the number after the second module names indicate the number of its output.
The directive **order** defines the execution order of the modules that belong to a space. This directive allows to coordinate the computation of the various modules, that is to compute a module only if all its inputs coming from the predecessor modules have already been computed. The axes referred by *YAi* (*YA1*

for the 1st axis and *YA2* for the second) are fixed and traversed in the mentioned order. These directives allow YAO to generate the global graph (in space and time) and the **order** directive gives a topological order of the graph.

8 Conclusion

The YAO software is dedicated to variational data assimilation in numerical models. It allows the user, dealing with a discrete numerical model representing a physical phenomenon, to describe the basic computation at each grid point. The user has to declare the space size and the number of time steps, specify the initial conditions and other parameters, present observations in space/time domain, define the cost function and the scenario chosen for its minimization, etc.. The YAO software then generates an executable program that enables to start data assimilation sessions. The current version of YAO allows to deal with real applications. We presented the modular graph concept which is the core of YAO and the algorithms that have been implemented in the YAO software. The modular graph structure opens up prospects for research and improvement. An analysis of the modular graph's structure allows to address the automatic generation of the topological order and the automatic parallelization of the presented algorithms.

References

1. Data Assimilation, Concepts and Methods. In: Training Course Notes of ECMWF, European Center for Medium-range Weather Forecasts. vol. 53., UK (1997)
2. Le Dimet, F.-X., Talagrand, O.: Variational Algorithms for Analysis and Assimilation of Meteorological Observations: Theoretical Aspects. J. Tellus, Series A, Dynamic Meteorology and Oceanography 38(2) (1986)
3. Louvel, S.: Etude d'un Algorithme d'Assimilation Variationnelle de Données à Contrainte Faible. Mise en Oeuvre sur le Modèle Océanique aux Equations Primitives MICOM (in French). PhD, Université de Toulouse III, France (1999)
4. Sportisse, B., Quélo, D.: Assimilation de Données. 1ère Partie : Eléments Théoriques (in French). Technical report, CEREA (2004)
5. Talagrand, O.: The Use of Adjoint Equations in Numerical Modelling of the Atmospheric Circulation. In Griewank, A., Corliss, G., eds.: Workshop of Automatic, Differentiation of Algorithms: Theory, Implementation and Applications, Breckenridge, Colorado, USA (1991)
6. Talagrand, O.: Assimilation of Observations, an Introduction. J. Meteorological Society Japan 75 191–209 (1997)
7. Hascoët, L., Pascual, V.: TAPENADE 2.1 User's Guide. Technical report, INRIA, France (2004), `http://www-sop.inria.fr/tropics/papers/Hascoet2004T2u.html`
8. Naumann, U., Utke, J., Wunsch, C., Hill, C., Heimbach, P., Fagan, M., Tallent, N., Strout, M.: Adjoint Code by Source Transformation with OpenAD/F. In Wesseling, P., Périaux, J., Oñate, E., eds.: Proceedings of the European Conference on Computational Fluid Dynamics (ECCOMAS CFD 2006), TU Delft (2006), `http://proceedings.fyper.com/eccomascfd2006/documents/35.pdf`

9. Giering, R., Kaminski, T.: Recipes for Adjoint Code Construction. ACM Transactions on Mathematical Software 24(4) 437–474 (1998), http://www.FastOpt.com/papers/racc.pdf
10. Fouilloux, A., Piacentini, A.: The PALM Project: MPMD Paradigm for an Oceanic Data Assimilation Software. vol. 1685 of LNCS. Springer, Berlin (1999)
11. Ide, K., Courtier, P., Ghil, M., Lorenc, A.: Unified Notation for Data Assimilation: Operational, Sequential and Variational. Special Issue J. Meteorological Society Japan 75, Data Assimilation in Meteorology and Oceanography: Theory and Practice, 181–189 (1997).
12. Fletcher, R.: Practical Methods of Optimization, second edn. New York, John Wiley (1987)
13. Courtier, P., Thépaut, J., Hollingsworth, A.: A Strategy for Operational Implementation of 4D-VAR Using an Incremental Approche. Q. J. R. Meteorological Society 120 1367–1387 (1994)
14. Louvel, S.: Implementation of a Dual Variational Algorithm for Assimilation of Synthetic Altimeter Data in the Oceanic Primitive Equation Model micom. J. Geophys. Res. 106(C5) 9199–9212 (2001)
15. Weaver, A., Deltel, C., Machu, E., Ricci, S., Daget, N.: A Multivariate Balance Operator for Variational Ocean Data Assimilation. Q. J. Royal Meteorological Society (2005)
16. Gilbert, J., Lemarchal, C.: Some Numerical Experiments with Variable-storage Quasi-newton Algorithms. Mathematical Programming 45 407–435 (1989), http://www-rocq.inria.fr/estime/modulopt/optimization-routines/m1qn3/m1qn3.html
17. Brajard, J., Jamet, C., Moulin, C., Thiria, S.: Use of a Neuro-variational Inversion for Retrieving Oceanic and Atmospheric Constituents from Satellite Ocean Colour Sensor: Application to Absorbing Aerosols. Neural Networks 19(2) 178–185 (2006)
18. Badran, F., Berrada, M., Brajard, J., Crépon, M., Sorror, C., Thiria, S., Hermand, J.P., Meyer, M., Perichon, L., Asch, M.: Inversion of Satellite Ocean Colour Imagery and Geoacoustic Characterization of Seabed Properties: Variational Data Inversion Using a Semi-automatic Adjoint Approach. J. of Marine Systems 69 126–136 (2008)
19. Hermand, J.P., Meyer, M., Asch, M., Berrada, M.: Adjoint-based Acoustic Inversion for the Physical Characterization of a Shallow Water Environment. J. Acoust. Soc. Am. 119(6) 3860–3871 (2006)
20. Kane, A., Thiria, S., Moulin, C.: Développement d'une Méthode d'Assimilation de Données in Situ dans une Version 1D du Modèle de Biogeochimie Marine PISCES (in French). Master's thesis, LSCE/IPSL, CEA-CNRS-UVSQ laboratories, France (2006)
21. YAO: Home Page http://www.locean-ipsl.upmc.fr/~yao/