Nouvelles fonctionnalités, évolutions et optimisations du logiciel l'assimilation variationnelle Yao

Guillaume Rosinosky

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

PARIS

MEMOIRE

présenté en vue d'obtenir

le DIPLOME D'INGENIEUR CNAM

SPECIALITE : Informatique

OPTION : Informatique, modélisation, optimisation

par

Guillaume ROSINOSKY

Nouvelles fonctionnalités, évolutions et optimisations du logiciel d'assimilation variationnelle Yao

Soutenu le 26 octobre 2012

JURY

PRESIDENT :	**M. Christophe PICOULEAU**	**Professeur - CNAM Paris**
MEMBRES :	**Mme Isabelle HERLIN**	**Directeur de recherche - INRIA**
	M. Eric GRESSIER	**Professeur - CNAM Paris**
	M. Julien BRAJARD	**Maître de conférences - UPMC**
	M. Fouad BADRAN	**Professeur – CNAM Paris**

Remerciements

Je voudrais tout d'abord remercier M. Fouad Badran, qui m'a proposé ce stage, m'a appuyé et m'a permis ainsi de travailler sur un sujet extrêmement intéressant.

Je souhaite également remercier toute l'équipe MMSA du laboratoire LOCEAN de l'université Pierre et Marie Curie (UPMC) et en particulier : Luigi Nardi pour ses explications et son expertise dans le fonctionnement de Yao, Julien Brajard, Sylvie Thiria, Abdou Kane, Anastase Alexandre Charantonis, et Simon Benavides pour leurs nombreuses explications dans le domaine statistique, pour les toutes les discussions intéressantes que nous avons eu, et pour tout le temps que je leur ai pris.

Je remercie aussi Mme Isabelle Herlin, M. Eric Gressier, et M. Christophe Picouleau, d'avoir accepté de faire partie du jury.

Je remercie enfin le laboratoire LOCEAN et son personnel pour son accueil.

Glossaire

profilage de code : analyse du comportement d'un programme à l'aide d'informations récupérées durant son fonctionnement, servant en général à déterminer les points d'optimisation en vue d'augmenter la vitesse ou de diminuer la consommation de mémoire.

wrapper : patron de conception (design pattern) permettant de convertir l'interface d'une classe en une autre, que le client attend (Wikipedia)

refactoring (*refactorisation*) : opération de maintenance du code informatique visant à retravailler le code source dans le but d'améliorer sa lisibilité, simplifier sa maintenance ou changer sa lisibilité.

Table des matières

Introduction

Mon stage s'est déroulé durant un an au laboratoire LOCEAN. Celui-ci est situé sur le campus de Jussieu (Paris VI). Ses activités concernent l'étude des processus de la variabilité océanique et de leurs interactions avec la variabilité climatique. L'équipe MMSA (Modélisation et Méthodes Statistiques Avancées) , où j'ai effectué mon stage, a les objectifs suivants :

- développement des aspects théoriques de la modélisation statistique
- application de méthodologies pour traiter la couleur de l'eau et l'upwelling sénégalo-mauritanien
- détermination de modèles directs et inverses de la salinité de surface
- développement du logiciel d'assimilation de données Yao
- assimilation des données couleurs de l'océan dans les modèles biogéochimiques
- coopération avec les pays du sud

Mon rôle a été d'ajouter des fonctionnalités au logiciel Yao, ainsi que d'effectuer des optimisations du code généré. Le logiciel Yao est un générateur de code pour faciliter l'assimilation variationnelle.

Je commencerai par décrire dans une première partie les principes généraux de l'assimilation variationnelle, pour ensuite décrire les fonctionnalités de Yao. Je poursuivrai par une description plus technique du générateur Yao et des programmes générés par celui-ci. Enfin, dans les deux dernières parties, je décrirai les modifications que j'ai apporté au logiciel, avant de conclure ce mémoire.

I Présentation du sujet

I.1 Simulation numérique

La simulation numérique est une technique permettant de reproduire virtuellement un phénomène réel complexe sur ordinateur, ce à l'aide de modèles numériques. Elle est utilisée dans de nombreux domaines, tant scientifiques, qu'industriels, ou économiques. Cette technique permet d'avoir une idée sur le comportement général d'un système sans obligatoirement effectuer des expérimentations.

Un modèle numérique est un ensemble de fonctions mathématiques décrivant un phénomène complexe. Si on représente un modèle comme une boite noire, on peut retrouver en entrée de celui-ci des paramètres de contrôle. Ceux-ci représentent l'état initial du système. Une itération de modèle permet d'obtenir les valeurs de sortie. La modification des paramètres d'entrée permet d'estimer le comportement du système dans d'autres cas de figure.

Toutefois, un modèle n'est jamais parfait. En effet, il correspond en tout premier lieu à une représentation théorique approchée de la réalité. C'est pourquoi il est important de rapprocher les sorties du modèles avec des observations réelles du système étudié. Avec la connaissance de l'état initial du système réel, celles-ci permettent de vérifier la cohérence d'un modèle vis à vis de la réalité, et permettent de l'ajuster et d'obtenir des données plus proches de la réalité.

Le problème de l'ajustement d'un modèle à la réalité n'est pas trivial. Le principe d'utiliser les observations pour faire évoluer le modèle est à la base de l'assimilation de données, comme nous le verrons dans la partie suivante.

I.2 L'assimilation de données

Cette partie est librement basée sur la revue des méthodes d'assimilation de Daget [1]. Selon ce dernier, « l'assimilation de données est […] l'ensemble des techniques statistiques qui permettent d'améliorer la connaissance de l'état d'un système à partir de sa connaissance théorique et des observations expérimentales ». Il s'agit alors de confronter les sorties du modèle aux mesures observées leur correspondant, afin d'ajuster les paramètres de contrôle du modèle numérique en rapprochant les sorties de celui-ci des mesures observées.

La première utilisation de méthodes utilisant les observations remonte au XVIIIème siècle avec les développements de l'astronomie. La méthode des moindres carrés, développée parallèlement par Legendre, Gauss et Laplace a permis ainsi de premières avancées dans le domaine. Les développements importants suivants dans cette discipline ont été réalisés au XXème siècle par Fisher, Wiener, et Kalman.

Il existe deux familles de méthodes d'assimilation de données : les méthodes séquentielles et les méthodes variationnelles. Les méthodes séquentielles traitent les observations au fur et à mesure de leur disponibilité. Elles s'appuient sur l'étude statistique des états du système afin de déterminer celui qui, statistiquement, est le plus adapté aux observations.

L'approche variationnelle traite le problème globalement, en minimisant une fonction « coût » qui mesure l'écart entre les mesures observées dans l'espace et le temps, et les sorties du modèle numérique.

L'assimilation séquentielle s'appuie uniquement sur les observations du passé. L'assimilation variationnelle s'appuie sur l'ensemble des observations, passées et futures et sert ainsi aux réanalyses.

Le logiciel Yao traite uniquement les méthodes d'assimilation variationnelle, c'est pourquoi nous présenterons uniquement dans ce mémoire ces méthodes uniquement.

I.2.1 Assimilation variationnelle

Il existe deux catégories de méthodes : le 3DVAR, permettant l'observation du phénomène dans un espace à une, deux ou trois dimensions. Le 4DVAR y rajoute la prise en compte de l'évolution des observations en fonction du temps.

I.2.1.1 Notations

Nous utiliserons dans ce mémoire les notations suivantes, reprises des écrits précédents concernant Yao :

- M : le modèle direct décrivant l'évolution entre deux pas de temps, notés t_i et t_{i+1},
- n : le nombre de pas de temps du modèle,
- $\boldsymbol{x}(t_0)$: le vecteur d'état initial d'entrée du modèle, de dimension N,
- $M_i(x(t_0))$: l'état du modèle au temps t_i, à partir du vecteur initial d'entrée,

– $\boldsymbol{M}(t_i, t_{i+1})$ ou $\boldsymbol{M_i}$: le linéaire tangent correspondant à la matrice jacobienne du modèle M calculé en $\boldsymbol{x}(t_i)$. Il est défini par la formule suivante :

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

– $\boldsymbol{M_i^T}$: l'adjoint du modèle M_i calculé en $\boldsymbol{x}(t_0)$ est la matrice transposée du linéaire tangent :

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

– $\boldsymbol{x^b}$: un vecteur d'ébauche (estimation à priori du vecteur $\boldsymbol{x}(t_0)$),

– $\boldsymbol{y^0}$: l'ensemble des observations. Le vecteur $\boldsymbol{y_i^0}$ correspond aux observations au temps t_i. Ce vecteur peut être vide dans le cas où il n'y aurait pas d'observations au pas de temps en question.

L'espace des observations disponibles n'a pas toujours la même granularité que celui du modèle. Par exemple, l'espace d'un modèle océanique peut concerner des points d'une taille de 1 kilomètre par 1 kilomètre, tandis que les observations récupérées peuvent concerner des points d'une taille de 100 mètres par 100 mètres. Il faut alors reconvertir les observations de manière à ce qu'elles soient utilisables dans le modèle en question. Nous avons donc besoin de définir ces opérateurs :

– H_i : l'opérateur d'observation permettant de calculer les variables d'observations y_i au temps t_i, à partir du vecteur d'état $\boldsymbol{x}(t_i)$.

– $\boldsymbol{H_i}$: le linéaire tangent de l'opérateur H_i calculé en $\boldsymbol{x}(t_i)$.

Ces opérateurs ne sont pas parfaits, il en résulte donc une erreur d'observation, notée ε_i. Elle est définie sur l'espace des observations, ce pour chaque pas de temps i. On considérera l'erreur d'observation comme un vecteur de variables aléatoire de moyenne nulle. Cette erreur est définie par la relation suivante :

$$\boldsymbol{y_i^0} = H_i(M_i(\boldsymbol{x}(t_0))) + \varepsilon_i$$

I.2.1.2 Principe général

Le principe du 4D-VAR passe par la minimisation d'une fonction coût J, représentée dans l'expression I.2.

$$J(\boldsymbol{x}(t_0)) = \frac{1}{2}\sum_{i=1}^{n}(\boldsymbol{y}_i - \boldsymbol{y}_i^0)^T \boldsymbol{R}_i^{-1}(\boldsymbol{y}_i - \boldsymbol{y}_i^0) + \frac{1}{2}(\boldsymbol{x}(t_0) - \boldsymbol{x}^b)^T \boldsymbol{B}^{-1}(\boldsymbol{x}(t_0) - \boldsymbol{x}^b) \quad \text{(I.2)}$$

On peut retrouver dans cette formule d'un coté l'écart entre les paramètres d'entrée et l'ébauche (pour le premier terme), et de l'autre l'écart entre les observations et les valeurs obtenues par le modèle (pour le second). Les matrices de variance-covariance $\boldsymbol{B}$ et $\boldsymbol{R}_k$ (avec k pour chaque pas de temps du modèle) permettent ici de pondérer chaque terme. Elles concernent respectivement l'erreur d'ébauche, et l'erreur d'observation.

Le principe est ici de retrouver les paramètres d'entrée correspondant à la valeur la plus faible possible de cette fonction. En effet, plus celle-ci sera faible, plus les paramètres d'entrée reconstitués et les valeurs calculées par le modèle en résultant seront proches des observations. La minimisation de cette fonction rapprochera donc les paramètres d'entrée de la meilleure solution.

En général, pour la minimisation de fonctions mathématiques, on utilise une méthode de descente de gradients. Celle-ci nécessite de calculer les dérivées partielles de la fonction étudiée.

Au niveau informatique, on dispose généralement d'un programme implémentant le modèle direct. Une estimation simple des dérivées partielles peut être réalisée en perturbant légèrement les entrées du modèle, comme le montre la formule I.3.

$$\frac{\delta J}{\delta x_i} \approx \frac{J(\boldsymbol{x} + \Delta x_i) - J(\boldsymbol{x})}{\Delta \boldsymbol{x_i}} \quad \text{(I.3)}$$

Cette méthode est donc relativement facile d'implémentation. Cependant celle-ci est beaucoup trop coûteuse pour pouvoir être utilisée dans des modèles réels. Une méthode plus efficace est ici nécessaire.

I.2.1.3 Algorithme standard : la méthode adjointe

La méthode adjointe nécessite l'implémentation d'un nouveau programme calculant l'adjoint du modèle direct. En effet, le calcul de l'adjoint nécessite tout d'abord d'évaluer l'expression I.4, représentant le gradient d'une fonction coût J par rapport aux entrées du système x, et aux sorties y.

$$\nabla_x J = \boldsymbol{G}_x^T \nabla_y J \quad \text{(I.4)}$$

La matrice G_x^T est ici la transposée de la matrice jacobienne, composée par les dérivées partielles de chaque pas de temps. L'ensemble des matrices jacobiennes du modèle correspond donc à l'adjoint de celui-ci. Il est nécessaire d'en calculer les valeurs. Le résultat de l'adjoint de la fonction coût (décrit dans la formule I.2) est représenté dans la formule I.5.

$$\begin{aligned} \nabla_{x(t_0)} J &= \boldsymbol{G}^{\boldsymbol{T}}_{\boldsymbol{x(t_0)}} \nabla_y J + \boldsymbol{B^{-1}}(\boldsymbol{x}(t_0) - \boldsymbol{x}^{\boldsymbol{b}}) \\ &= \sum_{i=1}^{n} \boldsymbol{M_i^T}(x(t_0))\, \boldsymbol{H_i^T}[\boldsymbol{R_i^{-1}}(\boldsymbol{y_i} - y_i^0)] + \boldsymbol{B^{-1}}(\boldsymbol{x}(t_0) - \boldsymbol{x_b}) \end{aligned} \tag{I.5}$$

Grâce aux gradients obtenus par cette méthode, il est possible de réajuster les paramètres de la fonction coût, d'une manière similaire à celle s'appuyant sur la perturbation des entrées du modèle. La figure I.1 en présente l'algorithme.

Figure I.1: algorithme général de l'assimilation variationnelle.

Ici une itération est représentée.

Cette méthode ne nécessite qu'une passe avant du modèle, suivie d'un calcul de l'adjoint de celui-ci pour effectuer une itération de minimisation. Ceci est beaucoup moins coûteux que la première méthode.

Toutefois, il y a deux points négatifs à retenir :

- l'implémentation informatique du programme adjoint d'un modèle direct est une tâche compliquée qui nécessite un investissement important,
- la minimisation peut poser des problèmes selon la nature du modèle. En effet, certaines fonctions sont difficilement minimisables.

Pour pallier à ce dernier point, une méthode alternative peut être employée, l'algorithme incrémental. Celle-ci est décrite dans la partie suivante.

I.2.1.4 Algorithme incrémental

Dans le cas de modèles complexes, la fonction coût peut être extrêmement non linéaire voire non continue, et la détermination du minimum de celle-ci peut devenir alors problématique. La méthode incrémentale permet d'éviter les minima locaux en linéarisant localement la fonction coût.

Le principe de cet algorithme est de procéder à des approximations successives de la fonction coût en s'appuyant sur le tangent linéaire du modèle (décrit dans la formule I.1). La minimisation peut alors s'effectuer sur une fonction de nature quadratique, et donc d'une manière bien plus aisée. En effet les fonctions quadratiques sont continues et ont un seul extremum.

Il faut garder à l'esprit que le tangent linéaire d'une fonction reste une approximation éloignée de la solution réelle. Après avoir obtenu le minimum de la fonction coût linéarisée, il convient donc de se repositionner sur la fonction coût réelle, avant d'estimer à nouveau le tangent linéaire. Ceci explique la nécessité d'une boucle externe dans cet algorithme, la boucle interne correspondant à la minimisation de l'approximation.

La figure I.2 décrit un exemple d'utilisation de cette méthode.

Figure I.2: algorithme incrémental de l'assimilation variationnelle
La courbe en trait plein correspond à la fonction coût $J(\boldsymbol{x}(t_0))$. Chacune des paraboles en pointillés correspond à la fonction $J[\delta\boldsymbol{x}]$ initialisée à chaque itération k de la boucle externe. La boucle interne réalise une descente de gradient sur la parabole correspondant à l'approximation. La boucle externe, quant à elle, permet d'avancer en se repositionnant sur la fonction $J(\boldsymbol{x}(t_0))$ (courbe pleine) et de faire une nouvelle approximation correspondant à la parabole suivante dans la minimisation.

I.3 Outils logiciels pour l'assimilation

Au niveau informatique, deux difficultés se présentent lors de la mise en place de méthodes d'assimilation variationnelle sur un modèle. La première correspond à l'obtention d'un programme permettant de calculer le modèle adjoint du modèle en question. Le second correspond au lancement d'une session d'assimilation variationnelle suivant un scénario choisi. En effet, de nombreux paramètres entrent en jeu lors d'une session d'assimilation, comme le réglage des paramètres d'entrée, des observations, la vérification des erreurs de précision, etc. Ces points peuvent être extrêmement fastidieux à mettre en place.

Au vu de ces deux difficultés, deux grandes familles de logiciels d'assimilation ont été développées.

Les différenciateurs automatiques permettent d'aborder la première difficulté. Selon Wikipedia [2], « la différentiation automatique est un ensemble de techniques servant à évaluer numériquement les dérivées de fonctions décrites par un programme informatique.

La différentiation s'appuie sur le fait qu'un programme utilise un ensemble séquentiel d'instructions élémentaires pour effectuer un calcul. Il est alors possible de calculer les dérivées de chacune de ces instructions afin d'obtenir la dérivée du modèle complet ». Des logiciels comme TAPENADE [3], TAF [4], TAC++ [5], et OpenDA [6] sont des différenciateurs automatiques de code. Ils permettent de générer le code du modèle adjoint à partir de celui du modèle direct.

La seconde famille correspond à des logiciels fournissant une panoplie d'outils permettant de gérer des scénarios d'assimilation. Des projets comme PALM [7] et OASIS [8], développés au CERFACS répondent à cette problématique. Pour cette famille de logiciels, l'utilisateur doit fournir le code du modèle direct, son adjoint, et éventuellement son tangent linéaire. Ce dernier peut être généré de manière automatique, par un différenciateur automatique par exemple.

Il existe plusieurs autres outils comme on peut le retrouver sur le site OpenDiff [9] développés dans de nombreux langages.

Chapelle *et al. [10]*, ont réalisé un comparatif entre plusieurs outils logiciels pour l'assimilation de données. Le tableau I.1 montre un récapitulatif des 5 logiciels présentés dans ce comparatif. On notera qu'on y retrouve également Yao. Ceci permet de le situer vis à vis des autres logiciels.

Outil	**Langage**	**Script**	**Interface graphique**	**Assimilation séquentielle**	**Assimilation variationnelle**	**Parallélisation**
DART [11]	Fortran 90	Shell	-	Oui	Non	Oui (en cours)
Oops	C++, F90	Python	-	?	Oui	Oui
OpenDA	C,C++, F77/F90, Java	XML	Oui	Oui	Non	Non
PALM	Appels à inclure dans code source	-	Oui	Oui	Oui	Oui
Yao	C++	Shell + Script Yao	En cours	Non	Oui	Oui (en cours)
Verdandi	C++	Python	Non	Oui	En cours	Oui (en cours)

Tableau I.1: Tableau comparatif de quelques outils d'assimilation

Le logiciel Yao, développé au LOCEAN, couvre les deux aspects décrits ci-dessus. Il est, d'une part, un générateur automatique de l'adjoint, et d'autre part, il inclut des fonctionnalités lui permettant d'agir comme une plate-forme pour lancer des scénarios d'assimilation de données. Yao n'est cependant pas un différenciateur automatique : il ne

génère pas le code adjoint à partir d'un programme implémentant le modèle direct. Il est basé sur une représentation du modèle numérique sous forme de graphe modulaire. Celui-ci correspond au flot de calcul décrit par le modèle.

Yao offre un langage de commande permettant à l'utilisateur de décrire les différents modules de calcul, l'échange de données entre eux, l'espace de discrétisation, la trajectoire temporelle, etc. A partir de ces informations YAO génère le graphe modulaire du modèle et applique des algorithmes de propagation et de rétropropagation dans ce graphe afin de calculer le modèle direct, le tangent linéaire, ainsi que le modèle adjoint. D'autres commandes permettent de lancer des sessions d'assimilation de données variationnelles 4D-VAR suivant des scénarios spécifiés par l'utilisateur.

II Description de YAO

Je vais présenter dans ce chapitre les principes généraux de Yao, le détail des fonctionnalités puis enfin quelques exemples de projet implémentés.

II.1 Principe général

Yao est un générateur de code pour l'assimilation variationnelle sous licence CECILL [12] dont le but est d'aider à la mise au point de modèles théoriques. Le langage de programmation utilisé pour le développement du générateur et des modules est le C++. On peut cependant noter qu'il est également possible de s'appuyer sur des modules développés en Fortran.

L'utilisateur doit avoir préalablement représenté son modèle sous la forme d'un graphe modulaire, où chaque module est une partie de l'ensemble des équations. Celles-ci sont ainsi retranscrites sous la forme de modules reliés entre eux. Les entrées et sorties d'un module peuvent être reliés aux sorties et aux entrées d'autres modules. Il est également possible d'y relier des initialisations du milieu extérieur, ou bien des paramètres de sortie du modèle. L'alimentation d'un module par un autre s'apparente à la composition de fonctions. Prenons comme exemple l'équation II.1.

$$y = \sin\left(x^2\right) \tag{II.1}$$

Cette formule simple représente la composition des deux opérations sinus et élevage au carré. Si on souhaite implémenter cette fonction dans un projet Yao, et que l'on souhaite séparer les deux opérations, on peut envisager un module sinus et un module carré. Ces deux modules auront chacun une entrée et une sortie. La sortie du module carré sera reliée à l'entrée du module sinus. L'entrée du module carré sera le paramètre de contrôle, et la sortie du module sinus, la sortie du modèle.

Pour le calcul du modèle adjoint ou du tangent linéaire, Yao nécessite l'implémentation des fonctions dérivées partielles des sorties par rapport aux entrées. Dans cet exemple, les fonction cosinus et $2x$ seront réalisées pour respectivement les modules sinus et carré.

Un exemple plus élaboré de graphe de liaisons entre modules peut être visualisé en figure II.1.

Figure II.1 : représentation d'un graphe modulaire d'application Yao.

La figure de gauche présente le détail des connexions entre modules, et la figure de droite une vue simplifiée. Quatre modules sont représentés ici, chacun ayant deux entrées et deux sorties. Les paramètres de contrôle sont incorporés au modèle par les modules F_p et F_l, par les entrées x_{p1}, x_{p2}, et x_{l2}. Les sorties du modèles, comparables à des observations sont les sorties y_{q1}, y_{r1} et y_{r2} des modules F_q et F_r.

Le calcul du modèle adjoint passe par la rétropropagation des valeurs depuis les sorties du modèle.

II.1.1 Concepts généraux de Yao

L'utilisateur doit décrire le modèle sous la forme d'un fichier de description, associé à des fichiers de modules contenant les calculs de la passe avant et du jacobien (matrice des dérivées partielles des sorties par rapport aux entrées d'une fonction).

Les modules que nous avons décrits dans la partie précédente représentent donc des unités de calcul (directive *module*), et sont reliés entre eux par des connexions (directive *ctin*). Dans la transcription d'un module Yao, l'utilisateur doit définir également les espaces sur lequel est défini le modèle (directive *espace*) -dont le nombre de dimensions et la taille sont réglables- et les trajectoires (directive *traj*) correspondant à la définition de la dimension temporelle du problème. Il est possible de déterminer le nombre de pas de temps, ainsi que la date de démarrage et la durée réelle d'un pas de temps. Ceci permet de définir des trajectoires de granularité différente et pouvant survenir simultanément.

Les modèles les plus complexes sont ainsi descriptibles en tant que projets Yao. Pour les besoins de l'assimilation, il est également nécessaire de spécifier si un module est destiné à recevoir des observations, et si le calcul de la fonction coût s'y applique.

Yao génère à partir de ces informations le graphe modulaire du problème, sur lequel il peut appliquer les opérations passe avant (*forward*), tangent linéaire (*linward*), et adjoint

(*backward*), qui doivent avoir été préalablement programmées par l'utilisateur pour chaque module.

Une relation entre deux modules (que l'on nommera *connexion*) concerne un pas de temps et un point de grille de l'espace discrétisé actuellement traité. Il est également possible de définir des connexions « absolues » pointant vers une coordonnée précise de l'espace. Afin d'illustrer ceci, la figure II.2 présente la représentation d'un modèle en 1 dimension, à laquelle se rajoute la dimension temporelle.

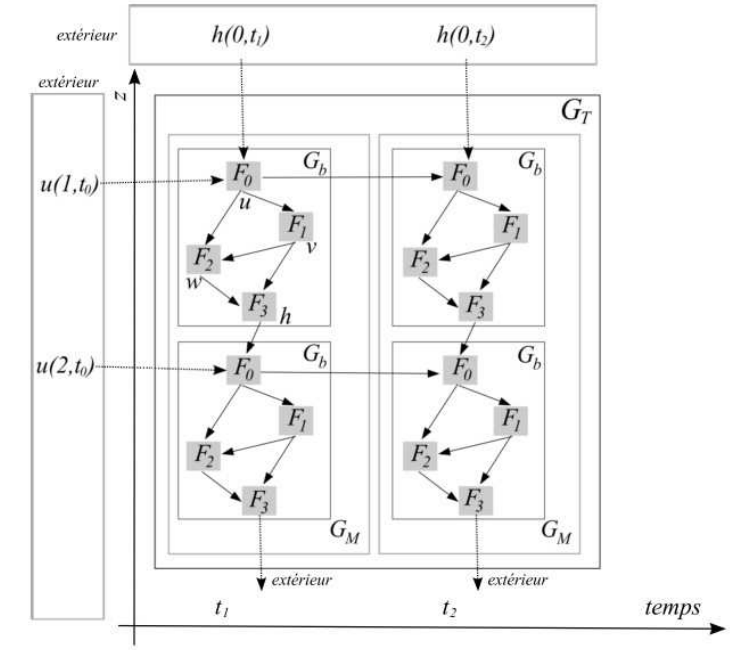

Figure II.2: exemple d'un graphe modulaire simple correspondant à un modèle direct défini sur un espace de dimension 1 et entre deux pas de temps *t-1* et *t*.

Figure reprise de la thèse de Luigi Nardi

Dans le graphe modulaire décrit par cette figure, on peut dire que le module F_0 au temps *t* est alimenté par par le module F_0 au temps $t-1$ et au même emplacement dans l'espace, mais également par le module F_3 au temps t et d'emplacement $z-1$ dans l'espace.

Les « bords » du modèle (conditions aux limites) doivent être également initialisés convenablement vis à vis du problème étudié. Ceci est réalisé en général dans le code source des procédures *forward* et *backward* des modules.

Un dernier point concerne l'ordre de calcul des modules. Actuellement, l'utilisateur peut spécifier celui-ci, ou demander à Yao sa détermination automatique. Les algorithmes ne sont pas triviaux, et sont décrits par Luigi Nardi dans sa thèse [13] ainsi que dans le mémoire d'ingénieur CNAM de Christophe Bernard [14].

II.2 Détail des fonctionnalités

Comme nous l'avons décrit dans la partie I.2.1.2, l'assimilation nécessite la minimisation d'une fonction coût. Deux méthodes de minimisation sont fournies à l'utilisateur par Yao : l'algorithme « standard » de minimisation s'appuyant sur une descente simple de gradient, et un autre s'appuyant sur une bibliothèque de minimisation. Ces deux méthodes seront décrites dans les parties suivantes. Nous présenterons ensuite succinctement les fonctions de tests proposées aux utilisateurs. Celles-ci servent avant tout à vérifier la bonne transcription du modèle dans Yao.

II.2.1 Algorithme « standard » de minimisation

Il s'agit ici de s'approcher du minimum de la fonction coût par une descente de gradient simple. On retire à la valeur des paramètres d'entrée cités dans la formule II.2 la valeur du gradient multipliée par une valeur epsilon choisie par l'utilisateur. Une nouvelle itération est alors lancée avec ces nouvelles valeurs, jusqu'à ce que le nombre d'itérations demandé par l'utilisateur soit atteint.

$$x_0 = x_0 - \varepsilon \delta x \qquad \text{(II.2)}$$

En général, cet algorithme garantit l'obtention du minimum global dans le cas où la fonction coût est convexe. En effet, dans le cas inverse, l'algorithme peut rester bloqué sur un minimum local non optimal. Le choix de la valeur ε est ici très important. En effet, dans le cas d'une valeur trop basse, la descente de la fonction coût se fera trop lentement. L'atteinte de l'optimum risquerait alors de nécessiter un nombre d'itérations (et donc un temps de calcul) trop élevé. Un ε trop élevé peut également provoquer un grand nombre d'itérations, l'optimum pouvant être alors dépassé voire même ne jamais être atteint.

II.2.2 Fonctionnement du minimiseur M1QN3

Le problème de la méthode « standard » est la gestion directe du pas de gradient ε. Pour pallier à ce problème, une bibliothèque développée par l'INRIA, le module m1qn3 [15], a été interfacée à Yao afin de fournir à l'utilisateur une méthode de minimisation supplémentaire. Cette procédure est adaptée aux problèmes de minimisation d'une fonction

f sans contraintes, et ayant un très grand nombre de variables. Elle utilise une méthode de type quasi-Newton. Son fonctionnement interne s'appuie sur une estimation de la matrice hessienne de la fonction étudiée. Elle comporte deux modes : le mode d'initialisation scalaire (SIS) et le mode d'initialisation diagonale (DIS). Elle nécessite le passage de plusieurs paramètres, dont en particulier :

- le vecteur d'entrée de la fonction étudiée,
- la valeur de la fonction coût,
- le gradient relatif à la valeur de la fonction coût,
- divers paramètres internes (comme le nombre de simulations, d'itérations, la précision attendue, etc.).

Il existe également son pendant « borné » M2QN1/N2QN1, permettant une minimisation avec contraintes fixes, c'est à dire permettant de retrouver des valeurs comprises dans un intervalle fixé au démarrage de la minimisation. Les conditions d'utilisation de cette procédure sont décrites dans la figure II.3.

Figure II.3: Conditions d'utilisation de la procédure m1qn3

(figure reprise du manuel m1qn3 [45])

Le principe général est de fournir à m1qn3 les paramètres décrits plus haut, ainsi qu'une pointeur vers une fonction simulateur (fonction *simulator* dans la figure) respectant une signature précise. Celle-ci doit calculer à partir des paramètres d'entrée fournis par le minimiseur la valeur de la fonction coût et les gradients de celle-ci. Ces paramètres sont modifiés en regard de la précédente itération, elle même basée sur la valeur de la fonction

coût et des gradients. m1qn3 peut avoir à tester plusieurs directions avant d'arriver à estimer convenablement le minimum. Cette fonction est appelée jusqu'à ce qu'un des critères d'arrêt soit atteints.

Une partie de mon travail a consisté à mettre en place la dernière version du minimiseur. La version de m1qn3 utilisée dans Yao à mon arrivée était la version *2.0e* et l'intégration de ce minimiseur dans le générateur posait des problèmes de licence pour une distribution de Yao. En effet, m1qn3 n'est sous licence GPL que depuis la version *3.2*, et il m'a été nécessaire de la mettre à jour. Ceci permet également de profiter des dernières avancées et optimisations apportées à celui-ci.

II.2.3 Algorithme avec minimiseur

Comme nous l'indiquons dans la partie précédente, à chaque simulation le minimiseur a besoin d'avoir en paramètre la valeur des paramètres d'entrée, la valeur de la fonction coût calculée par une passe avant, et le gradient des paramètres d'entrée par rapport à la valeur de la fonction coût, calculée par une rétropropagation.

Tous les modules déclarés avec le mot clé « *target* » (représentant les paramètres d'entrée du modèle) sont utilisés dans la minimisation. Ainsi les tableaux fournis au minimiseur lors d'une itération doivent contenir l'ensemble des états (et des gradients) de chaque module l'un après l'autre. Il est également possible d'effectuer une minimisation sur plusieurs pas de temps simultanément.

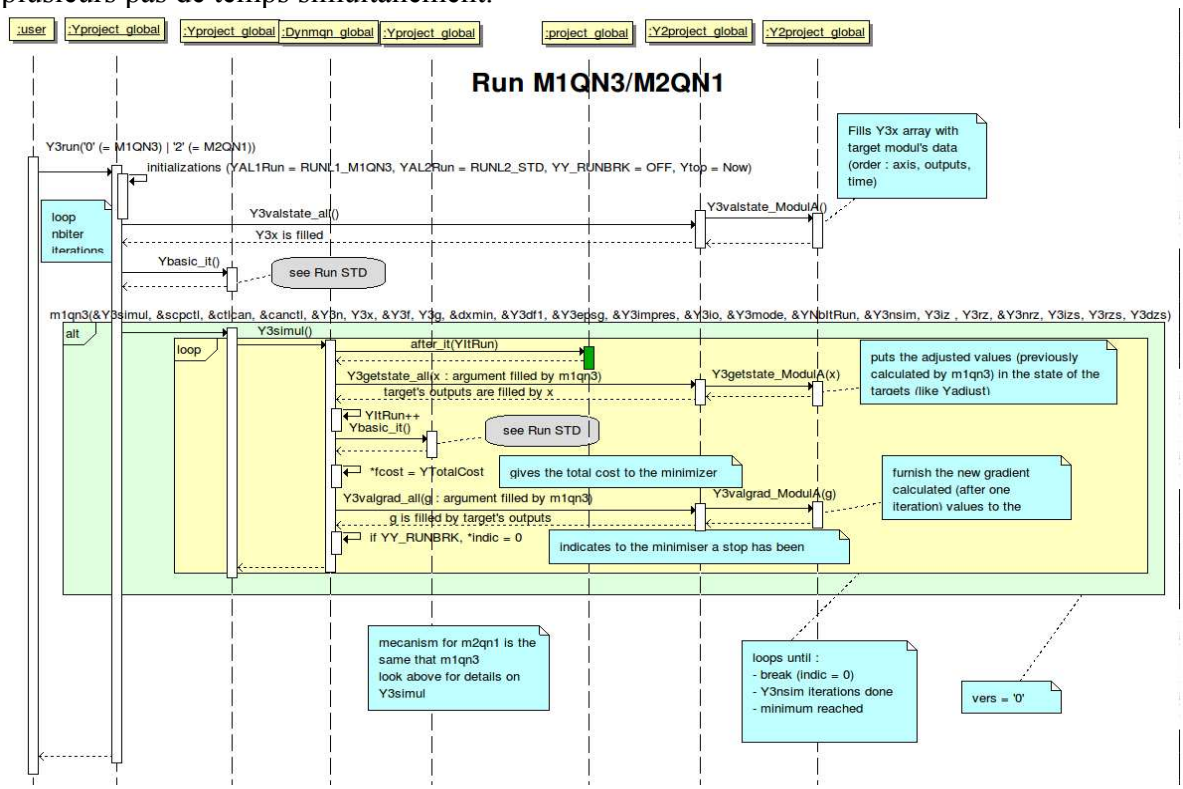

Figure II.4: Diagramme de séquence de l'appel au minimiseur

Dans la figure II.4, on peut retrouver un diagramme de séquence que j'ai réalisé, repris de la documentation technique de Yao. Celui-ci montre l'algorithme général avec minimiseur. On remarque bien ici que la fonction *m1qn3* appelle la fonction *Y3simul* plusieurs fois.

Comme il a été précisé dans la partie précédente, le minimiseur attend une fonction de « simulation ». Il s'agit de la fonction *Y3simul* dans le schéma, qui lance un *forward* suivi d'un *backward* afin d'obtenir la valeur de la fonction coût et les gradients. En effet, dans son comportement par défaut[1], m1qn3 effectue lui-même les boucles de minimisation. La fonction de simulation récupère ensuite les valeurs de x fournies par le minimiseur (*Y3getstate_all*), déclenche une itération, et met à jour les gradients et la valeur de la fonction coût dans les tableaux fournis au minimiseur.

II.2.4 Fonctions additionnelles de tests

Pour vérifier que le modèle et son inverse ont été convenablement implémentés, il existe un ensemble de fonctions d'aide à la mise au point fournies par Yao. Je vais présenter succinctement ces quatre méthodes dans cette partie. Cette partie s'appuie sur le manuel Yao et sur la partie concernant ces tests de la thèse d'Abdou Kane [16].

Le premier test présenté ici est la fonction *testdf*. Son but est de vérifier que les fonctions servant à calculer les jacobiens sont exactes. Par exemple, pour un module effectuant l'opération $y = x^2$, la fonction backward correspondante doit fournir le résultat $y = 2x$. Pour cela, Yao s'appuie sur la dérivée fonctionnelle (dite de Fréchet) et la dérivée directionnelle (dite de Gateaux). Le fonctionnement de ces dérivées s'apparente à celle décrite par la formule II.2. Si les différences entre les valeurs obtenues par Yao et le test n'excèdent pas la précision fournie en entrée, le test est considéré comme concluant. Le deuxième test (fonction *testob*) vérifie les deux dérivées précédemment décrites sur la fonction coût.

Il existe également le test de l'adjoint, et le test du linéaire tangent permettant de vérifier l'exactitude de la transposition du tangent linéaire. On peut retenir que ces tests permettent de vérifier que les connexions entre les modules sont convenablement réalisées.

[1] Le mode actuellement utilisé par Yao est le mode direct (*direct mode*). Depuis la version 3.2 de m1qn3, il est possible de configurer le minimiseur pour fonctionner en mode inverse (*reverse mode*). Dans ce mode, celui-ci fournit à la fonction appelante les nouveaux paramètres d'entrée, et indique si les conditions d'arrêt ont été atteintes. Ce dernier laisse effet une plus grande liberté au programme, avec un fonctionnement plus simple. Il permettrait également d'apporter de meilleures généralisations aux algorithmes de Yao.

II.3 Exemple de projects implémentés sur YAO

Nous présentons ici quelques modèles implémentés dans Yao, en particulier le modèle du Shallow Water.

II.3.1 Shallow Water

Ce modèle, appelé également modèle de Saint Venant, provient de l'intégration verticale du modèle de Navier-Strokes à 3 dimensions. Le but de celui-ci est de d'écrire l'écoulement d'un liquide non visqueux, en eau peu profonde, et à surface libre. Il est basé sur l'ensemble d'équations différentielles suivant :

$$\frac{\delta u}{\delta t} = -g^* \frac{\delta h}{\delta x} + f v + \gamma u \tag{II.3}$$

$$\frac{\delta v}{\delta t} = -g^* \frac{\delta h}{\delta y} + f u + \gamma v \tag{II.4}$$

$$\frac{\delta h}{\delta t} = -H\left(\frac{\delta u}{\delta x} + \frac{\delta v}{\delta y}\right) \tag{II.5}$$

On retrouve dans ces équations les termes suivants :

- u et v sont les vitesses sur les axes x et y (respectivement),
- g^* représente la gravité réduite,
- h est la hauteur de la surface libre,
- γ est un facteur de dissipation,
- H est la hauteur d'eau moyenne.

Ce modèle est discrétisé grâce à la grille C d'Arakawa pour l'espace, et à l'aide d'une discrétisation de type Leap Frog couplée à un filtre d'Asselin pour la dimension temporelle. Les équations résultantes sont les suivantes :

$$u_{ijt} = \overline{u}_{ijt-2} + 2\Delta t\left(\frac{-g^*}{\Delta x}\left[h_{i+1\,j\,t-1} - h_{ijt-1}\right] + \frac{f}{4}\left[v_{ijt-1} + v_{i+1\,j\,t-1} + v_{i+1\,j+1\,t-1}\right] - \gamma\,\overline{u}_{ijt-2}\right) \tag{II.6}$$

$$v_{ijt} = \overline{v}_{ijt-2} + 2\Delta t\left(\frac{-g^*}{\Delta y}\left[h_{i\,j\,t-1} - h_{i\,j-1\,t-1}\right] - \frac{f}{4}\left[u_{i-1\,j-1\,t-1} + u_{i-1\,j-1\,t-1} + u_{i\,j\,t-1}\right] - \gamma\,\overline{v}_{ijt-2}\right) \tag{II.7}$$

$$h_{ijt} = \overline{h}_{i\,j\,t-2} - 2\Delta t\,\dot{H}\left(\frac{u_{i\,j\,t-1} - u_{i-1\,j\,t-1}}{\Delta x} + \frac{v_{i\,j+1\,t-1} - v_{i\,j\,t-1}}{\Delta y}\right) \tag{II.8}$$

$$\overline{u}_{t\,j\,t} = u_{i\,j\,t} + \alpha\left(\overline{u}_{t\,j\,t-1} - 2\,u_{i\,j\,t} + u_{i\,j\,t+1}\right) \tag{II.9}$$

$$\overline{v}_{ijt} = v_{ijt} + \alpha\left(\overline{v}_{ijt-1} - 2\,v_{ijt} + v_{ijt+1}\right) \quad \text{(II.10)}$$

$$\overline{h}_{ijt} = h_{ijt} + \alpha\left(\overline{h}_{ijt-1} - 2\,h_{ijt} + h_{ijt+1}\right) \quad \text{(II.11)}$$

Dans la représentation Yao de ce modèle, il y a 6 modules reliés entre eux, chacun reprenant une des équations précédentes (u, $\overline{u}$, v, $\overline{v}$, h, $\overline{h}$). Ce modèle est présenté plus précisément dans le rapport de recherche concernant Yao [17]. Il s'agit de l'un des premiers modèles implémentés sur Yao, et, pour être précis, de celui utilisé en général pour l'ajout de nouvelles fonctionnalités. Les résultats de l'expérience jumelle décrite dans le rapport de recherche sont édifiants. L'erreur obtenue est extrêmement faible, sans utiliser d'ébauche.

II.3.2 Modèle d'infiltration

Ce modèle, basé sur le modèle Seshiba est actuellement en cours de portage sur Yao par Simon Benavides, qui effectue sa thèse au laboratoire LOCEAN. Il concerne l'infiltration de l'eau dans les sols. Il sera décrit plus avant dans la partie IV.1.2.1 concernant le test de l'implémentation de l'assimilation quasi-statique.

II.3.3 Autres modèles

Il existe d'autres modèles ayant été implémentés sur Yao :

- le modèle de transfert radiatif Neurovaria [18], concernant l'inversion des reflectances satellitaires pour contrôler des paramètres atmosphériques et océaniques,
- le modèle de biogéochimie PISCES [16], pour l'assimilation des concentrations en nanophytoplancton, dans le but de contrôler des paramètres de production de mortalité et de broutage,
- le modèle d'océnographie NEMO [19] dans sa configuration Gyre,
- etc.

III Structure logicielle

Dans cette partie, je vais tout d'abord présenter la structure d'origine du générateur, et d'un programme généré. Les parties suivantes décriront les évolutions structurelles réalisées sur le générateur et sur l'ensemble des programmes générés par ce dernier.

III.1 Structure d'un programme généré

On peut retrouver dans la figure suivante l'organisation des différents fichiers utilisés. Il est possible de regrouper ces derniers en quatre blocs : les fichiers écrits par l'utilisateur (*user project files*), les fichiers générés (*generated files*), les fichiers fournis par Yao (*YAO headers and script files*), et les bibliothèques externes (*external libraries*). Ces quatre blocs seront décrits dans les parties suivantes.

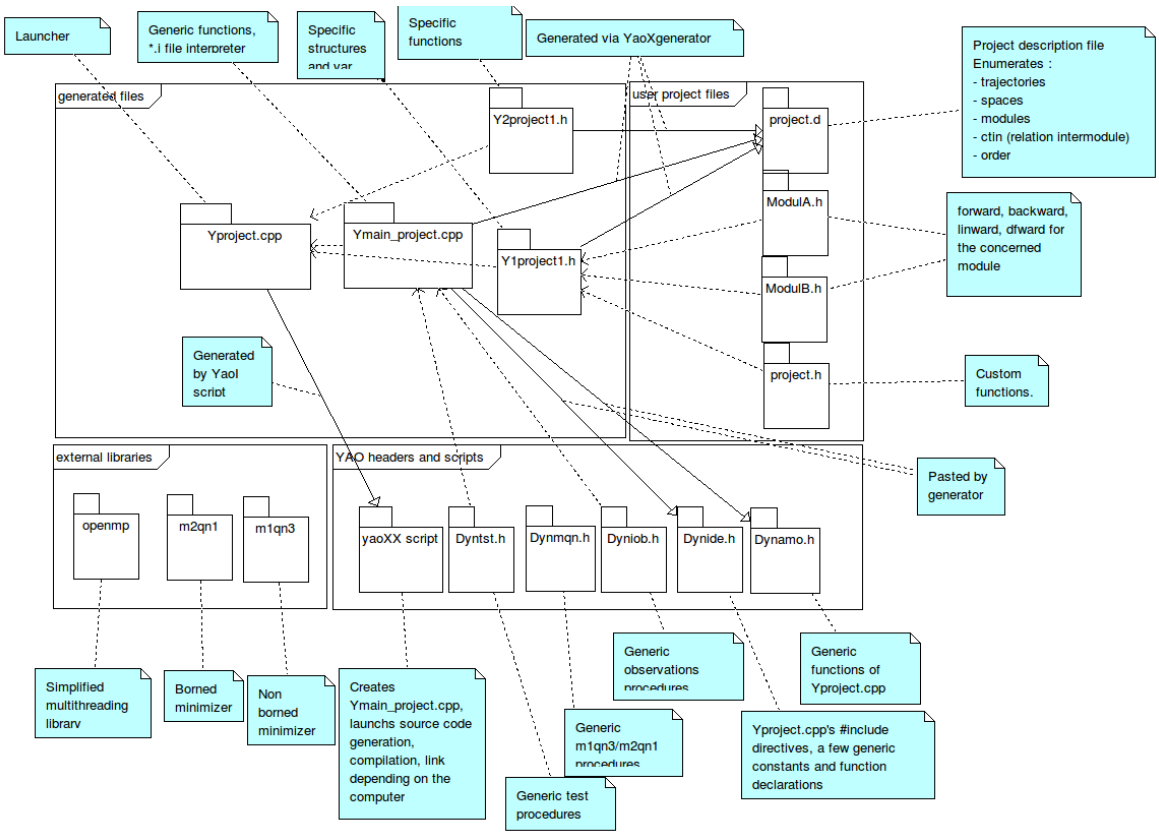

Figure III.1: structure d'un programme généré

III.1.1 Fichiers écrits par l'utilisateur

Le fichier *description* de ce bloc (*project.d* dans la figure III.1) contient la description de l'ensemble des modules, espaces, trajectoires, ainsi que les liens les reliant.

Il contient également les options destinées à être incorporées au projet, comme par exemple l'utilisation d'un minimiseur particulier, la prise en charge de la parallélisation, etc. Nous présentons ci-après un exemple de fichier de description correspondant au projet du Shallow Water présenté dans le paragraphe II.3.1.

```
defval SZX 50                                                    ┐
defval SZY 50                                                    │ définition de
defval SZU 1                                                     │ variables
defval SZT 50                                                    │
defval SZA 51                                                    ┘

option o_m1qn3                                                   ─ option : choix du minimiseur

traj   Toce    M  SZU    SZT                                     ─ définition d'une trajectoire

space   Soce       M   SZX SZY      Toce                         ─ définition d'un espace

modul Hfil space Soce input 3 output 1 tempo cout target         ┐
modul Ufil space Soce input 3 output 1 tempo                     │
modul Vfil space Soce input 3 output 1 tempo                     ─ définition des modules du projet
modul Hphy space Soce input 5 output 1 tempo                     │
modul Uphy space Soce input 7 output 1 tempo                     │
modul Vphy space Soce 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-1  j    t-1                          │
ctin Hphy 3 from Uphy  1  i    j    t-1                          │
ctin Hphy 4 from Vphy  1  i    j    t-1                          │
ctin Hphy 5 from Vphy  1  i    j+1  t-1                          ─ connexions entres modules
                                                                 │
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 modinspace Soce                                            ┐
          order YA1 YA2                                          │
                Hphy Uphy Vphy                                   │
                Hfil Ufil Vfil                                   │
          forder                                                 ─ ordre de traitement des calculs
forder                                                           │
                                                                 │
order spaceintraj Toce                                           │
          Soce                                                   │
forder                                                           ┘
```

Figure III.2: Exemple de fichier de description (Shallow Water)

Dans l'exemple de fichier de description présenté dans la figure III.2, le lecteur retrouvera la déclaration d'une trajectoire (de nom *Toce*), d'un espace (de nom *Soce*), des 6 modules composant le projet (respectivement *Hfil*, *Hphy*, *Ufil*, *Uphy*, *Vfil*, *Vphy*), de leurs connexions (représentées par les directives *ctin*), et de l'ordre de traitement de ces modules

(représenté par les directives *order*). Concernant la syntaxe des connexions, la première ligne avec le mot clé *ctin* signifie que la première sortie du module *Hfil* situé au point de grille (i,j) du temps (t-1) alimente ce même module dans sa première entrée (étant sous entendu que l'alimentation se fait à l'emplacement i,j et au pas de temps courant).

L'utilisateur doit également avoir préparé chaque module sous la forme d'un fichier en-tête C contenant le code source des fonctions de la passe avant du module et de son jacobien. La signature de la fonction contient l'ensemble des entrées du module.

La fonction *forward* correspond à la passe avant du module. Chaque paramètre correspond à une entrée du module (soit une sortie d'un autre module). Les sorties du module sont représentées sous la forme de variables dont le nom est YS*a* où *a* est le numéro de la sortie, déduit de la formation du fichier description. Les figures III.3 et III.4 représentent les fonctions *forward* et *backward* du module *Hfil.*

```
forward (YREAL Entree1, YREAL Entree2, YREAL Entree3)
{
            if (Yt <= 2)
                        YS1 = Entree1;
            else
                        YS1 = Entree2 + alpha*(Entree1 - 2* Entree2 + Entree3);
}
```

Figure III.3: Fonction de calcul de la passe avant du module Hfil

La figure III.4 représente la fonction *backward* du module Hfil.

```
backward (YREAL Entree, YREAL Entree2, YREAL Entree3)
{
            if (Yt <= 2)
                        YJ1I1 = 1.;
            else
            {
                        YJ1I1 = alpha;
                        YJ1I2 = 1 - 2 * alpha;
                        YJ1I3 = alpha;
            }
}
```

Figure III.4: Fonction de calcul du jacobien du module Hfil

La fonction *backward* correspond au calcul du jacobien du module, utilisée pour la rétroprogagation du gradient et pour le calcul du tangent linéaire décrits dans la partie I.2.1.1. Elle prend en entrée les sorties résultantes d'une passe avant précédemment effectuée. Elle fournit en sortie le jacobien de la fonction. De la même manière que pour les variables YS*a* de la passe avant, les variables représentant le jacobien sont Yj*a*I*b.* Chaque composante correspond à la dérivée partielle de la sortie *a* par rapport à l'entrée *b*. Dans le cas présenté, comme nous avons trois éléments composant le jacobien, au vu du fait que ce module a 3 entrées et 1 sortie.

Les variables YS*a* et YJ*a*I*b* sont respectivement des directives de précompilation pointant sur le tableau des sorties d'un module, et sur le tableau des jacobiens (celui-ci étant réutilisé).

L'utilisateur doit décrire le fichier *project*.i correspondant à l'ensemble des directives de l'assimilation. Il a ainsi accès à un ensemble de fonctions permettant par exemple de charger des observations, d'initialiser le nombre d'itérations d'un algorithme et de le lancer. Un interpréteur de fichier de script Yao est inclus en standard dans les fichiers générés, comme nous le verrons dans la partie suivante.

Il existe également un fichier que nous nommerons ici *project*.h, contenant un ensemble de procédures personnalisables par l'utilisateur et lancées par le générateur à des moments bien précis comme avant ou après une itération, avant ou après une passe avant, au démarrage du programme (pour initialiser des variables spécifiques). Par exemple on peut dénoter la fonction *forward_after*() appelée après une passe avant. On peut noter qu'il est possible d'ajouter des fonctions à la grammaire de l'interpréteur, en les entrant dans ce fichier, et en les signalant au niveau du fichier description.

III.1.2 Fichiers générés par Yao

Les deux fichiers Y1*project*.h et Y2*project*.h, contiennent respectivement la déclaration de l'ensemble des variables et structures spécifiques nécessaires au projet ainsi que l'ensemble des traitements spécifiques nécessaires au projet. Ces fichiers sont déduits du fichier description, et font explicitement référence aux fichiers source des modules créés par l'utilisateur.

Le fichier Y*project*.cpp contient l'interpréteur de fichier .i, ainsi que les fonctions génériques d'appel ne nécessitant pas de génération spécifique. Il s'agit du fichier principal faisant appel à l'ensemble des modules. Le fichier Ymain_project.cpp est celui qui contient la fonction *main* du programme, soit son point d'entrée. Il fait appel directement à une fonction main dérivée dans le fichier Y*project*.cpp. L'utilité de ce fichier est donc limitée. Ces fichier sont reconsistués et créés par le script shell *YaoI* à partir des fichiers Ydynamo.cpp, Ydynamo.h, Ydynide.h (décrits dans la partie suivante).

III.1.3 Fichiers fournis par Yao

Les traitements génériques sont pour la plupart dans un répertoire spécifique (*$YAODIR/yao/include/*), ajouté à la compilation du projet. Les fichiers présents permettent la gestion des observations, de la minimisation par M1QN3, des tests, ainsi que les 2 fichiers permettant la génération du fichier Y*project*.cpp :

- *dynamo.h*, *dynamo.cpp* et *dynide.h* servent à la consistitution du fichier «principal » contenant l'interpréteur et incluant les autres fichiers sources ;
- *dynmqn.h* contient la plupart des fonctions de minimisation ;
- *dyntst.h* contient les fonctions liés aux tests ;
- *dynobs.h* contient les fonctions d'entrée sortie liées aux observations, ainsi que les fonctions liées aux calculs de la fonction coût.

Le script shell *YaoI* a un rôle primordial : celui-ci lance le générateur, crée les fichiers contenant l'interpréteur (*Yproject.cpp*) à partir de ses 3 fichiers constituants tout en rajoutant les lignes nécessaires. Il crée également celui contenant la fonction main(*Ymain_project.cpp*). Après toutes ces opérations il lance la compilation du projet.

III.1.4 Bibliothèques nécessaires pour un projet Yao

Il existe également des composants optionnels d'un projet Yao : les bibliothèques Openmp [20], m1qn3 [15], et m2qn1 [21]. Selon les options indiquées dans le fichier description, celles-ci seront liées au projet final, et le code correspondant à leur utilisation généré.

M1QN3 et M2QN1 ont été décrits dans la partie II.2.2. Il existe des fonctions spécifiques utilisant ces minimiseurs dans un programme généré par Yao. L'interpréteur fournit à l'utilisateur la possibilité de lancer une assimilation avec minimiseur, ainsi qu'un ensemble d'instructions pour régler ce dernier. Comme nous l'avons vu dans la partie précédente, il existe un fichier fourni par Yao s'occupant de la plupart des traitements liés au minimiseur, mais certaines structures supplémentaires spécifiques au projet sont également ajoutées, celles-ci étant ajoutées par la générateur si l'option est ajoutée dans le fichier description.

Openmp est une API permettant la parallélisation dite à mémoire partagée d'un programme. Ceci signifie qu'elle est réalisée pour les plateformes utilisant une même mémoire. Elle fonctionne par la création et l'utilisation de plusieurs threads en parallèle (multithreading). Elle est multiplateforme et relativement simple d'utilisation. Son utilisation par Yao a été ajoutée par Luigi Nardi lors de sa thèse [13]. L'utilisation de la parallélisation permet d'améliorer les performances en effectuant simultanément certains calculs des modules. Le principe est ici de marquer les boucles de traitement des espaces comme étant parallélisables. Ces calculs devant pouvoir être effectués simultanément, il ne

doit donc pas y avoir de relations entre les points de grilles calculés sous peine d'erreurs au niveau de l'écriture ou de la lecture. Ceci nécessite donc au niveau du générateur de vérifier les dépendances entre modules et de marquer les boucles parallélisables. Certaines boucles ne peuvent être parallélisées à cause de ces dépendances.

Les directives utilisées sont ajoutées au niveau des boucles de traitement des espaces (celles-ci étant déclarées comme parallélisables), et au niveau de certaines variables qu'il n'est pas envisageable d'accéder en même temps.

III.2 Structure du générateur

La structure du générateur implémentée au départ par Charles Sorror, a été modifiée par Luigi Nardi durant ses travaux au LOCEAN. Il a implémenté la structure objet présentée sous la forme d'un digramme de classe simplifié dans la figure III.5.

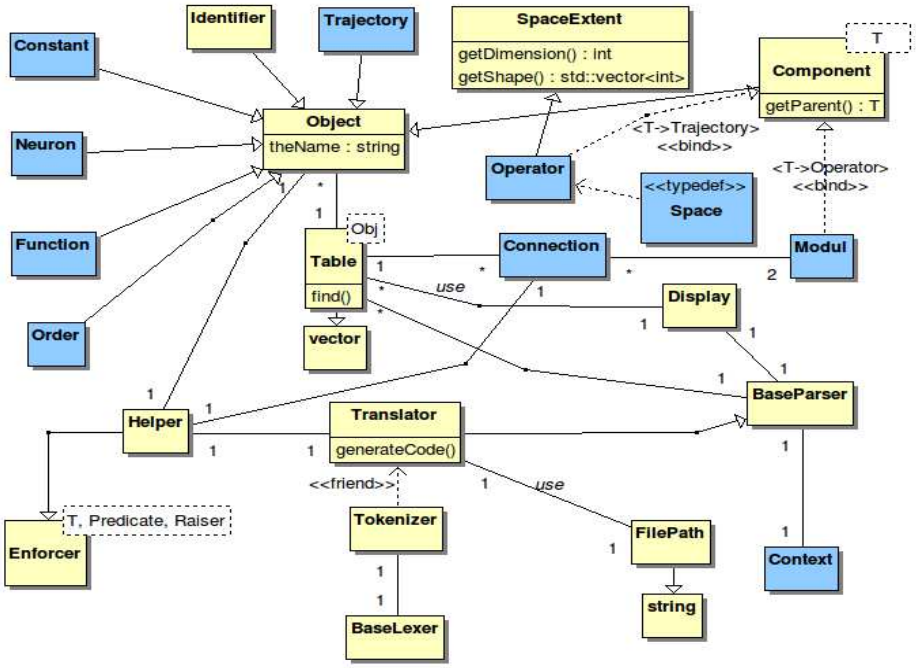

Figure III.5: diagramme de classe simplifié du générateur Yao

Les classes *Tokenizer*, *BaseLexer* et *BaseParser* sont générées à partir d'un fichier destiné à être interprété par Antlr (*grammar.g*). Ce fichier contient la grammaire du fichier de description. On peut y retrouver la liste des mots clés, que du code source servant à remplir les listes d'éléments, ainsi que le retour d'erreurs à l'utilisateur. La classe *Translator* contient l'ensemble de la génération de source. Elle s'appuie sur des listes de vecteurs des classes dérivées de la classe Object remplies par le traitement précédemment décrit. Ces

classes correspondent aux entités que l'on peut retrouver dans le fichier description. Les listes sont traitées les une après les autres, le générateur créant le code source nécessaire tant dans le fichier de déclaration (*Y1project.h*) que dans celui de traitements (*Y2project.h*).

Le générateur s'appuie sur les bibliothèques suivantes :

- antlr [22]: Another Tool for Language Recognition est un framework libre pour la mise au point de compilateurs. Il propose une analyse de type LL(*) [23] et est utilisé pour la génération de l'interpréteur de fichier description.
- Boost Graph Library [24] et [25]: Boost est un ensemble libre de bibliothèques C++. La Boost Graph Library concerne la gestion des graphes et les algorithmes les concernant. Le générateur utilise cette bibliothèque pour le contrôle de cohérence l'ordre des modules, pour la génération automatique de celui-ci, ainsi que pour la parallélisation.

La première action réalisée lors du lancement du générateur est l'analyse du fichier description écrit par l'utilisateur. Cette analyse remplit les listes d'éléments précédemment citées. Le générateur fait ensuite appel à des fonctions dédiées pour chaque type de liste (trajectoire, module, etc.). Ces fonctions remplissent au fur et à mesure le fichier de déclaration, et le fichier d'implémentation. Les fichiers Y1project.h et Y2project.h décrits dans la partie III.1.2 sont ainsi créés par le générateur.

Les classes BaseParser et BaseLexer sont générées par Antlr à l'aide de la grammaire codée dans le fichier *grammar.g*. Elles remplissent les listes d'objets (trajectoire, espace, module, etc.) suite à l'interprétation du fichier description. Ces classes effectuent également un test sur le respect des règles de grammaire de ce fichier. Les fonctions GenerateCode (voir schéma) du générateur créent ensuite les fichiers du projet selon la liste des éléments chargée en mémoire.

III.3 Restructuration des programmes générés Yao

Comme il est précisé dans la partie précédente, le générateur a été restructuré. Cependant les programmes générés ont gardé sensiblement la même structure depuis le début. Aucune restructuration complète du code n'a été réalisée, et les fonctionnalités et le code correspondant ont été introduits au fur et à mesure. Ceci provoque un manque de lisibilité au niveau du code généré par Yao, et à terme un manque de portabilité.

La restructuration partielle que j'ai réalisé a consisté a effectuer un refactoring du code Yao généré. J'ai en l'occurrence ajouté un ensemble de classes, tout en gardant la structure précédente. En effet, vis à vis du temps disponible un refactoring complet de l'ensemble des fonctions de Yao n'était pas envisageable. J'ai ainsi repris certaines structures existantes et les ai formalisées sous forme de classes, ceci permettant ainsi de bénéficier de concepts objet comme l'héritage ou le polymorphisme. La nouvelle structure des projets Yao est décrite dans la figure III.6.

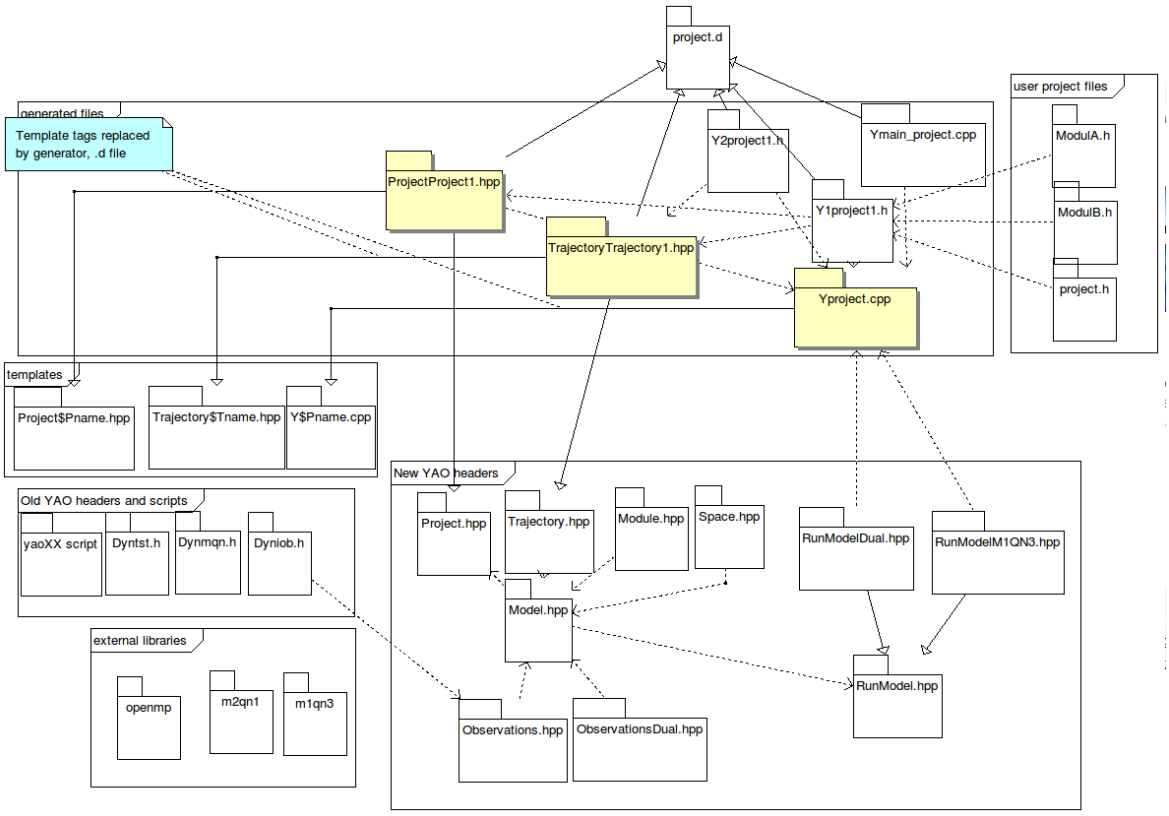

Figure III.6: Nouvelle structure d'un programme généré

III.3.1 Nouveaux fichier inclus

J'ai représenté les concepts principaux de Yao sous la forme de classes. Voici une description succinte de chacunes d'entre elles :

- L'entité module (classe *Module*) représente la vue générique d'un module. L'ensemble des données (états des modules, gradients, etc.) est toujours stocké dans les variables précédentes (localisées dans le fichier *Y1project.h*), ce afin de garder la compatibilité ascendante avec les anciens projets Yao.

- L'entité trajectoire (classe *Trajectory*) est ici une classe abstraite. Les classes dérivées correspondant à chaque trajectoire définie par l'utilisateur sont générées automatiquement par Yao. Ceci était important car les boucles de traitement des modules sont désormais à ce niveau[2], et il était donc nécessaire d'avoir une classe pour chaque trajectoire réelle.
- L'entité espace (classe *Space*) décrit un espace et est reliée à une trajectoire.

En plus de ces concepts, j'ai ajouté les entités suivantes :

- une entité modèle (classe *Model*) pour contenir l'ensemble des espaces, trajectoires et données, ainsi que les traitements génériques les concernant. On peut citer comme traitement les accesseurs vers les différentes entités contenues, ainsi que les fonctions concernant la gestion du temps.
- une entité abstraite « parcours de modèle » (classe *RunModel*), contenant les traitements offerts par Yao (par exemple, passe avant de modèle, passe arrière, etc.). Les entités dérivées correspondent aux traitements spécifiques assimilés aux commandes *run*. En l'occurrence, on peut citer le parcours de l'assimilation standard, de l'assimilation avec minimiseur, de l'assimilation incrémentale. On peut également retrouver l'assimilation duale, décrite dans le chapitre IV.
- une entité abstraite (classe *Project*) correspondant au projet, et contenant le modèle. Une classe héritée correspond au projet en lui même. Il contient en particulier les initialisations.
- un ensemble de classes (classes *Observations*, *ObservationsDual*) correspondant à la gestion générique des observations (valable seulement pour l'algorithme dual à ce jour). Celles-ci seront décrites dans la partie IV.2.11.

[2] Nous avons vu précédemment que les boucles de traitement étaient découpées auparavant par espace. Les raisons de ce changement seront précisées dans la partie V.4.

L'ensemble de ces entités est décrit dans la figure III.7.

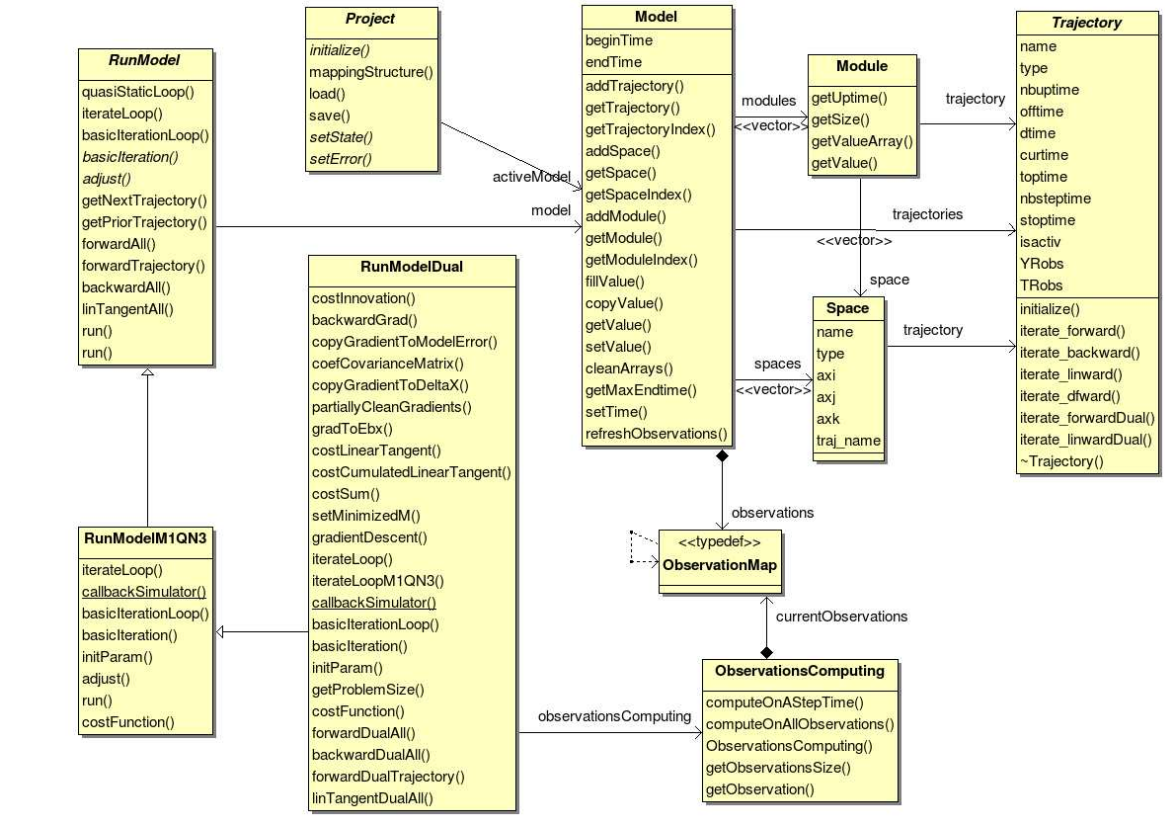

Figure III.7: Diagramme de classe des concepts de la nouvelle structure

III.3.2 Fichiers template

Pour l'implémentation des nouvelles fonctionnalités, il m'est apparu que dans la génération de code source nécessaire, il existait une grande quantité d'éléments fixes, ceux-ci représentant un canevas. En effet, les nouvelles classes projet et trajectoire faisant appel au polymorphisme, les déclaration des classes héritées et de leur fonctions restent identiques. Seul le nom de la classe et certains traitements changent selon le fichier généré. Il pouvait donc être intéressant de mettre au point un système de canevas, le générateur s'occupant simplement de remplacer des métadonnées par le code source spécifique nécessaire. Un autre atout à l'utilisation de ce type de système est qu'il permet une modification plus aisée de ceux-ci et qu'il devient possible pour un utilisateur avancé de les modifier, comme nous le verrons dans la partie suivante.

Les fichiers canevas utilisés sont les suivants :

- Project*$pname*.hpp, est le canevas du fichier de la classe projet héritée. Il contient les initialisations spécifiques du programme (modules, trajectoires, etc.)
- Trajectory*$Tname*.hpp, est le canevas des fichiers des classes trajectoires héritées. Pour chaque trajectoire, il en résultera un fichier correspondant, contenant une classe spécifique à celle-ci.
- Y*$Pname*.cpp est le canevas du fichier source Y*$project*.cpp contenant les fonctions de l'interpréteur du projet. Ce fichier remplace les fichiers Dynamo.cpp, Dynamo.h et Dynide.h en en reprenant le contenu. Ceci simplifie grandement la structure des fichiers nécessaires à la génération d'un projet, tout en replacant la responsabilité de cette génération dans le générateur, en lieu et place du script YaoI.

La variable *$pname* est remplacée par le nom du projet. La variable *$tname* est remplacée par le nom de la trajectoire concernée.

III.4 Modifications et fonctionnalités supplémentaires du générateur

Nous allons examiner ensuite les autres modifications apportées au générateur dans le cadre de la restructuration du code source du projet.

III.4.1 Ajout d'un système de patrons de fichiers générés

Comme je l'ai évoqué dans la partie précédente, les fichiers « templates » présentés dans la figure III.6 possèdent des variables destinées à être remplacées. Les fichiers générés (projet, trajectoires, et Y*project*.cpp) sont tous les trois destinés a accueillir du source générés, tout en ayant besoin de source fixe. Les fichiers Y1*project.h* et Y2*project.h* sont entièrement générés et sont destinés à terme à ne plus être utilisés, j'ai donc gardé la génération précédente pour ces derniers.

Il existe des bibliothèques de remplacement de code source, comme par exemple autogen [26], mais vu que mon temps était limité, que mes besoins de génération de code n'étaient pas trop complexes et que je ne souhaitais pas mettre en place une solution trop compliquée, j'ai préféré coder ceci « à la main ».

J'ai ajouté à l'ensemble des classes présentées dans le schéma III.6 la classe *TemplateAnalyzer*. Celle-ci prend en entrée le fichier canevas nécessaire, le nom du fichier destiné, et l'instance de la classe *Translator* appelante (pour permettre un accès à

l'ensemble des listes d'entités interprétées). La figure III.8 présente un diagramme de séquence représentant la dynamique d'appel de ces classes.

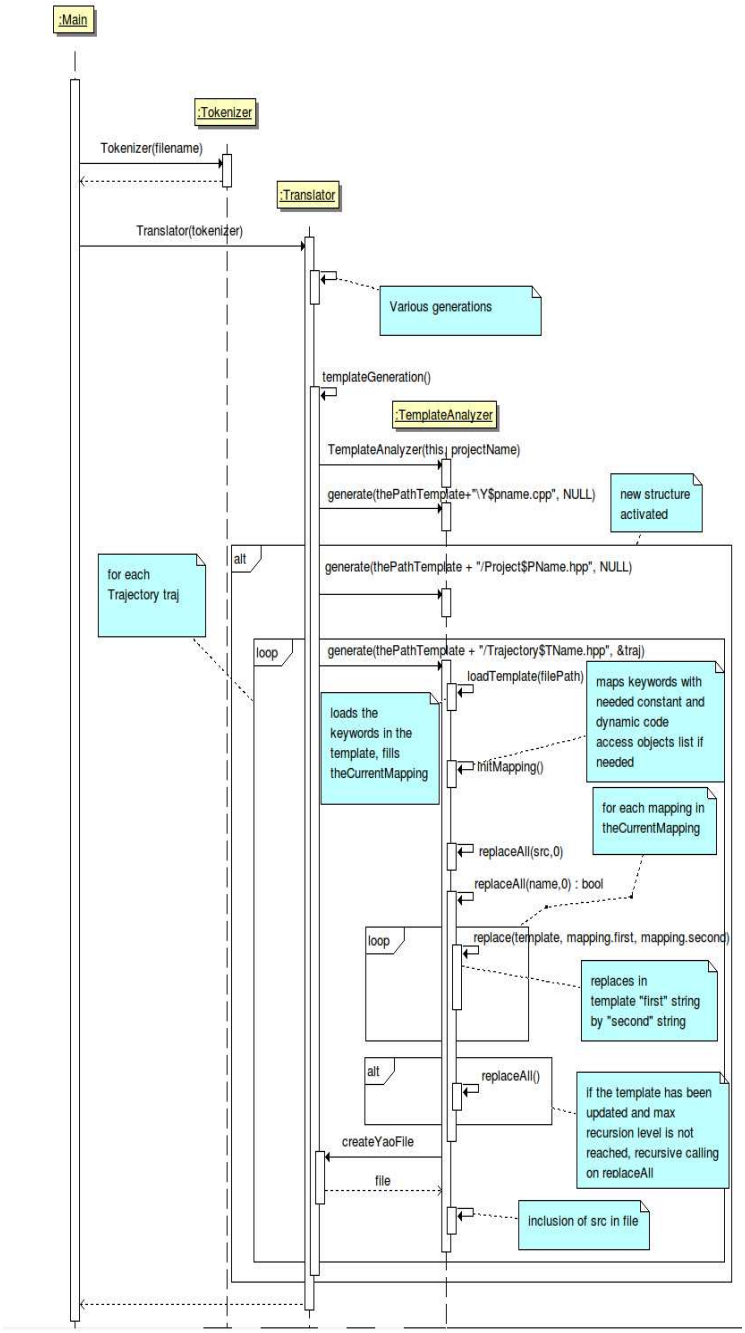

Figure III.8: Diagramme de séquence relatif à la génération des fichiers de la nouvelle structure

Comme l'indique la figure III.8, l'appel à la classe de génération de canevas nécessite que l'ensemble des structures aient été remplies, et que le source nécessaire ait été généré.

III.4.2 Autres modifications du générateur

Pour ajouter ces nouvelles fonctionnalités, il m'a été nécessaire d'effectuer d'autres modifications au niveau du code source du générateur. En l'occurrence, les boucles de traitement des espaces, tout en étant toujours ajoutées dans le fichier d'implémentation (Y2project.h) sont également ajoutées dans un flux (de classe *stringstream*) repris tel quel dans la partie de la génération du canevas.

J'ai ajouté au générateur une option permettant de ne pas générer le projet selon la nouvelle structure, ce pour garder la compatibilité ascendante, et en cas de problème non remarqué aux tests sur la nouvelle structure. Il faut cependant noter que l'ancienne structure ne permet pas d'utiliser les nouvelles fonctionnalités qui seront décrites dans le chapitre IV. La génération à l'aide de la classe *TemplateAnalyzer* est dans ce cas utilisée seulement pour le fichier Y*project*.cpp.

Une autre fonctionnalité que j'ai ajouté est la possibilité pour l'utilisateur de choisir le répertoire contenant les fichiers « template » et celui contenant les fichiers inclus. Ceci lui permet d'utiliser s'il le souhaite une version alternative des template ou des fichiers inclus. Cette possibilité est évidemment réservée aux utilisateurs ayant une très bonne connaissance de la structure d'un programme généré. L'utilisateur doit également garder à l'esprit que ces fichiers peuvent être modifiés avec une nouvelle version du générateur, et qu'il aura peut être un travail de vérification et de portage à réaliser lors des mises à jour du générateur.

IV Implémentation de nouvelles fonctionnalités

Hormis la partie optimisation, ma tâche a concerné également l'ajout de deux nouvelles fonctionnalités à Yao : l'assimilation quasi-statique et la méthode duale.

IV.1 Assimilation quasi-statique

IV.1.1 Théorie

Le principe de l'assimilation quasi-statique (QSVA) a été développée par Pires et al. [27]. L'appelation « progressive variationnal assimilation » : assimilation variationnelle progressive est également employée comme l'indique Ferron [28]. Le but premier de la méthode était de passer outre les problèmes d'assimilation survenant dans les modèles hautement chaotiques.

Son principe est de commencer par assimiler les données sur une durée temporelle courte correspondant à une fraction de la durée totale, et en partant des résultats obtenus, de lancer des assimilations sur des durées croissantes en reprenant les résultats, ce jusqu'à la durée totale de l'expérience.

Comme l'indique Rouston [29], le principe de l'assimilation quasi-statique est le suivant, avec λ une portion de la durée totale d'assimilation, et T la durée sur laquelle l'assimilation est lancée :

1. Démarrer à partir d'un first-guess, et initialiser $T = \lambda$
2. Lancer l'assimilation sur la durée T, et garder le résultat obtenu comme état du système au temps t_0
3. Prolonger T de λ, répéter la phase 2 jusqu'à ce que T atteigne la durée totale d'assimilation.

IV.1.2 Implémentation de l'assimilation quasi statique

Cette méthode requiert la connaissance des deux paramètres suivants :

- Un premier paramètre qui précise le début effectif de l'assimilation. Ce terme, utilisable également hors assimilation quasi-statique indique à quelle valeur de temps réel le « run » doit commencer. En effet, on peut prévoir, comme le montre le schéma ci dessous, que pour un projet donné les premières trajectoires ne servent qu'à l'initialisation.

– Le nombre d'intervalles qui partitionnent la durée globale. Pour cette évolution, je suis parti du postulat que la durée d'assimilation est découpée en partie égales en temps absolu. Il sera possible d'ajouter ultérieurement des commandes permettant de définir précisément des durées d'assimilation.

J'ai tout d'abord noté qu'il s'agit d'une évolution sans impact sur la grammaire du fichier de description. En effet, il est préférable que l'utilisateur puisse changer les réglages nécessaires dynamiquement à l'intérieur du fichier interprété. Cela signifie que nous allons modifier la procédure de gestion de l'interpréteur, en ajoutant les deux commandes nécessaires. L'assimilation quasi statique peut s'appliquer à tous les types de Run existant actuellement.

On peut voir cette évolution comme le rajout d'une boucle extérieure aux algorithmes généraux, avec une modification au niveau du début et de la durée (en temps réel) de l'assimilation. J'en ai profité pour commencer à mettre en place la nouvelle structure des projets, ce de manière partielle, en implémentant seulement les classes Project et Trajectory. Ceci a provoqué une quantité de modifications relativement faible permettant d'éviter une modification trop lourde immédiate.

Le diagramme de séquence de la figure IV.1 représente le comportement implémenté.

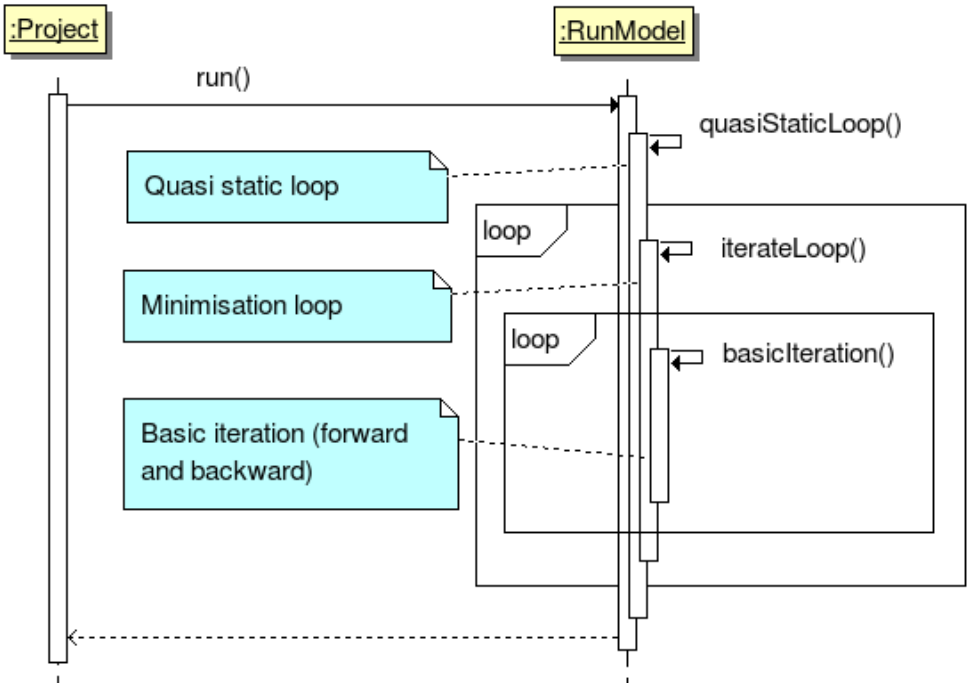

Figure IV.1 Diagramme de séquence de l'implémentation de l'algorithme quasi-statique

La fonction *quasiStaticLoop* détermine la durée d'assimilation par rapport au nombre d'itération défini par l'utilisateur. Les itérations d'assimilation nécessaires sont alors lancées via les fonction *iterateLoop* (lancant la boucle de minimisation) et *basicIteration* (lancant une itération, soit une passe avant suivie d'une rétropropagation).

Ce comportement fait désormais totalement partie des programmes générés, et est implémenté dans les classes abstraites. Ceci signifie que l'algorithme quasi-statique peut être utilisé avec tous les algorithmes implémentés au niveau des classes *RunModel*.

IV.1.2.1 Tests et résultats

Le modèle testé est le modèle d'infiltration des sols décrit succintement dans la partie II.3.3.

Le modèle décrit la pénétration de l'eau dans le sol. Ce processus découle de plusieurs paramètres comme ses caractéristiques hydrodynamiques, sa texture et sa structure, ainsi que les conditions initiales et le flot d'arrivée d'eau. Selon les conditions, les différentes couches terrestres peuvent être saturées d'eau et un écoulement peut avoir lieu. Le cycle de l'eau est complexe, et de nombreux phénomènes peuvent survenir comme des précipitations, de l'évaporation, des écoulements, de l'infiltration, de la percolation, du stockage et des écoulements souterrains.

Il s'agit ici d'un modèle simplifié, avec une seule dimension spatiale, et une dimension temporelle. La dimension spatiale s'étend depuis la surface, correspond à différentes couches sédimentaires et s'arrête sur une couche infranchissable.

Les paramètres du contrôle du modèle sont les suivants :

- coefficient de conductivité de saturation
- pression relative de saturation
- saturation en eau de la couche
- paramètre de structure du sol
- quantité d'eau initiale du sol
- quantité d'eau initiale en surface
- quantité d'eau initiale du fond

Les résultats obtenus suite à l'apprentissage sont décrits dans le tableau IV.1.

	Champ d'essai	Assimilation classique	Assimilation quasi-statique en 5 segments
Erreur quadratique	0,1724	0,1527	0,1149

Tableau IV.1.1 Résultats obtenus lors de l'expérience du quasi statique sur le modèle d'infilration

On remarque bien ici une amélioration des résultats obtenus dans l'assimilation quasi statique.

IV.2 Implémentation du 4DVar à contrainte faible (méthode duale)

La méthode de calcul utilisée actuellement dans YAO ne tient pas compte des erreurs du modèle. En effet, ce dernier n'est pas une représentation parfaite de la réalité, et contient donc des erreurs liées à cette imperfection. L'implémentation de l'algorithme à contrainte faible ajoute la prise en compte de ce biais et permet ainsi d'estimer la pertinence du modèle.

Suite à une présentation du problème et des rappels sur la programmation quadratique, nous verrons dans cette partie l'analyse, la conception et la mise en oeuvre de cette fonctionnalité. Le lecteur se référera au chapitre I pour retrouver les détails généraux sur l'assimilation variationnelle, ainsi que des explications plus précises concernant les diverses variables et concepts utilisés dans cette partie.

IV.2.1 Présentation du problème

IV.2.1.1 Introduction

Soit un modèle M prenant des paramètres x sur $\mathbb{R}^N$ avec des sorties sur $\mathbb{R}^N$ avec N correspondant au nombre de points de grille du modèle. Le modèle M peut aussi varier en fonciton du temps. C'est le cas lorsqu'il dépend de paramètres externes (comme par exemple la pluie dans les modèles hydrologiques). On notera alors par M_i le modèle entre deux pas de temps t_i et t_{i+1}.

Soient :

- n : le nombre de pas de temps du modèle,
- p_i : le nombre d'observations au pas de temps t_i ($1 \leq i \leq n$),
- x^b : le vecteur d'ébauche correspondant aux variables de contrôle,

- x_i^t : le vecteur (défini sur $\mathbb{R}^n$ représentant l'état réel du phénomène étudié au temps t_i)
- y_i^{obs} : les observations au temps t_i,

  Les observations y_i^{obs} sont définies en fonction de x_i^t par la formule suivante :

  $y_i^{obs} = H_i[x_i^t] + \varepsilon_i$ où :

- H_i est un opérateur d'observation permettant de retrouver un format correspondant aux observations y_i^{obs} à partir des paramètres x_i^t
- ε_i est l'erreur de mesure relative à l'opérateur d'observation H_i au temps t_i

On retiendra par la suite l'équation suivante, où M_i correspond à l'application du modèle au temps i :

$$x_i^t = M_{i-1}(x_{i-1}^t) + \eta_{i-1} \tag{IV.1}$$

Cette équation introduit le vecteur η_i qui correspond à l'erreur du modèle au pas de temps t_i. On supposera pour la suite que les vecteurs η_i sont les réalisations de variables multidimensionnelles aléatoires indépendantes. La variable η_i suit une loi normale de moyenne de vecteur $\vec{0}$, et de matrice de variance-covariance Q_i

Ici, le vecteur η_i correspond à l'erreur de représentabilité du modèle définie pour chaque pas de temps t_i. On considérera cette erreur comme un bruit blanc non prédictible indépendant du temps.

On peut représenter la fonction objectif de ce problème de la façon suivante :

$$J[x(t_0), \eta_0, \eta_1, \ldots, \eta_{n-1}] =$$

$$\frac{1}{2}[x_0^t - x^b]^T B_0^{-1}[x_0^t - x^b] \tag{IV.2}$$

$$+ \frac{1}{2}\sum_{i=1}^{n}[H(x_i^t) - y_i^{obs}]^T R_i^{-1}[H(x_i^t) - y_i^{obs}] \tag{IV.3}$$

$$+ \frac{1}{2}\sum_{i=1}^{n} \eta_{i-1}^T Q_i^{-1} \eta_{i-1} \tag{IV.4}$$

On remarquera que nous rajoutons à la fonction coût standard de l'assimilation variationnelle, le terme IV.4, relatif à l'erreur du modèle. Rappelons que notre objectif est de minimiser J relativement à x_t^0 et aux vecteurs η_i afin de parvenir à retrouver x_0^t proche de la réalité, et d'évaluer les vecteurs η_i qui corrigeront l'évolution du modèle M .

Pour rappel :

- B_0 est la matrice de variance/covariance sur l'erreur d'ébauche (background).
- R_i est l'ensemble des matrices de variance/covariance sur les erreurs d'observations, ce pour chaque pas de temps t_i .
- Q_i est l'ensemble des matrices de variance/covariance d'erreur sur le modèle, ce pour chaque pas de temps t_i .

Le but de la première partie est de représenter chaque membre de la fonction coût sous forme matricielle (en utilisant des matrices par blocs), ce afin de simplifier les calculs.

IV.2.1.2 Linéarisation et écriture du problème sous forme matricielle

Nous allons commencer par remplacer le terme concernant l'ébauche, que l'on retrouvera dans la formule IV.2. Pour la suite, nous retiendrons $\delta x_0 = x_0^t - x^b$, de manière à obtenir l'expression IV.5.

$$\frac{1}{2}\delta x_0^T B_0^{-1} \delta x_0 \qquad \text{(IV.5)}$$

Nous avons pour le pas de temps 1 :

$$x_1^t = M_0[x_0^t] + \eta_0$$

Que nous pouvons approximer de la manière suivante :

$x_1^t \simeq M_0[x^b] + \boldsymbol{M}_0 \delta_x + \eta_0$ avec $\boldsymbol{M}_i$ la matrice jacobienne du modèle au pas de temps t_i calculé en x_i^t (voir méthode du tangent linéaire, introduite dans le chapitre I), elle est de taille N x N (où N est la taille de l'espace du modèle).

Similairement, pour le deuxième pas de temps :

$$x_2^t \simeq M_1[x_1^t] + \eta_1$$

$$x_2^t \simeq M_{\ 1}[M_{\ 0}[x^b] + \boldsymbol{M}_0 \delta x_0 + \eta_0] + \eta_1$$
$$= M_{\ 1} M_{\ 0}[x^b] + \boldsymbol{M}_1 \boldsymbol{M}_0 \delta x_0 + \boldsymbol{M}_1 \eta_0 + \eta_1$$

Ceci est généralisable de la manière suivante :

$$\begin{aligned} x_i^t &\simeq M_{i-1} \circ M_{i-2} \circ \ldots \circ M_0(x^b) \\ &+ \boldsymbol{M}_{i-1} \boldsymbol{M}_{i-2} \ldots \boldsymbol{M}_0 \delta x_0 + \sum_{k=1}^{i-1} \boldsymbol{M}_{i-1} \boldsymbol{M}_{i-2} \ldots \boldsymbol{M}_k \eta_{k-1} + \eta_{i-1} \end{aligned} \qquad \text{(IV.6)}$$

La première ligne de l'équation IV.6 correspond au modèle appliqué sur i pas de temps avec le vecteur d'ébauche comme paramètre initial. La deuxième ligne représente le tangent linéaire où l'on retrouve l'erreur de représentativité du modèle. Celle-ci découle des approximations successives par le tangent linéaire.

On utilise ici la matrice $\boldsymbol{H}_i$, matrice jacobienne de l'opérateur d'observation. La taille de cette matrice est de $n \times p$ où p est le nombre de points d'observations.

Ainsi, après approximation par le tangent linéaire :

$$\begin{aligned} H_i(x_i^t) &\simeq H_i \circ M_{\ i-1} \circ M_{\ i-2} \circ \ldots \circ M_{\ 0}(x^b) \\ &+ \boldsymbol{H}_i (\boldsymbol{M}_{i-1} \boldsymbol{M}_{i-2} \ldots \boldsymbol{M}_0 \delta x_0 + \sum_{k=1}^{i-1} \boldsymbol{M}_{i-1} \boldsymbol{M}_{i-2} \ldots \boldsymbol{M}_k \eta_{k-1} + \eta_{i-1}) \end{aligned}$$

On en tire que :

$$\begin{aligned} H_i(x_i^t) - y_i^{obs} &\simeq \boldsymbol{H}_i \boldsymbol{M}_{i-1} \boldsymbol{M}_{i-2} \ldots \boldsymbol{M}_0 \delta x_0 + \sum_{k=1}^{i-1} \boldsymbol{H}_i \boldsymbol{M}_{i-1} \boldsymbol{M}_{i-2} \ldots \boldsymbol{M}_k \eta_{k-1} + \boldsymbol{H}_i \eta_{i-1} \\ &- [y_i^{obs} - H_i \circ M_{i-1} \circ M_{i-2} \circ \ldots M_0(x^b)] \end{aligned} \qquad \text{(IV.7)}$$

Nous allons maintenant représenter le terme concernant les erreurs sur les observations.

Soit le vecteur z_i de dimension p_i, défini par l'équation suivante, correspondant au premier membre de l'équation IV.7 et défini par :

$$z_i = \boldsymbol{H}_i \boldsymbol{M}_{i-1} \boldsymbol{M}_{i-2} \ldots \boldsymbol{M}_0 \delta x_0 + \sum_{k=1}^{i-1} \boldsymbol{H}_i \boldsymbol{M}_{i-1} \boldsymbol{M}_{i-2} \ldots \boldsymbol{M}_k \eta_{k-1} + \boldsymbol{H}_i \eta_{i-1} \qquad \text{(IV.8)}$$

Soit d_i le vecteur d'innovation défini par :

$$d_i = y_i^{obs} - H_i M_{i-1} \circ M_{i-2} \circ \ldots \circ M_0(x^b) \quad \text{(IV.9)}$$

L'équation IV.7 s'exprime alors par : $z_i - d_i = H\,[x(t_i)] - y_i^{obs}$, où $x(t_i) = M_{i-1} \circ M_{i-2} \circ \ldots \circ M_0(x_b)$.

On remplace $H\,[x(t_i)] - y_i^{obs}$ par $z_i - d_i$ dans l'équation IV.3, pour obtenir le terme IV.10 :

$$\frac{1}{2}\sum_{i=1}^{n} [z_i - d_i]^T \boldsymbol{R_i^{-1}} [z_i - d_i] \quad \text{(IV.10)}$$

IV.2.1.3 Écriture matricielle

Soit p le nombre total d'observations, défini par :

$$p = \sum_{i=1}^{n} p_i$$

On définit la matrice $\boldsymbol{R}$ par blocs, en disposant les matrices $\boldsymbol{R}_i$ au niveau de la diagonale, ce pour chaque vecteur observation. Cette matrice est donc de taille $p \times p$, et est définie par :

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_1 & & & \\ & \boldsymbol{R}_2 & & \\ & & \ldots & \\ & & & \boldsymbol{R}_p \end{bmatrix}$$

On pose les vecteurs définis par blocs $\boldsymbol{z}$ et $[\boldsymbol{z} - \boldsymbol{d}]$ de dimension p tels que $\boldsymbol{z}^T = (z_1^T, z_2^T, \ldots, z_n^T)$ et $[\boldsymbol{z} - \boldsymbol{d}]^T = [(z_1 - d_1)^T, (z_2 - d_2)^T, \ldots, (z_n - d_n)^T]$.

On définit ensuite la matrice $\boldsymbol{Q}$ par blocs, en disposant les matrices $\boldsymbol{Q_i}$ au niveau des diagonales, ce pour chaque pas de temps. Cette matrice est donc de taille $nN \times nN$ où n est le nombre de pas de temps du modèle et N la taille de l'espace de celui-ci.

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{Q}_1 & & & \\ & \boldsymbol{Q}_2 & & \\ & & \ldots & \\ & & & \boldsymbol{Q}_n \end{bmatrix}$$

On pose $\boldsymbol{\eta}$ le vecteur de dimension nN défini par $\boldsymbol{\eta}^T = [\eta_0^T, \eta_1^T, \ldots, \eta_{n-1}^T]$.

La fonction coût s'écrit sous la forme matricielle :

$$\begin{aligned} &J(\delta x, \eta, z) = \\ &\frac{1}{2}\delta x_0^T \boldsymbol{B_0^{-1}} \delta x_0 + \\ &\frac{1}{2}[\boldsymbol{z}-\boldsymbol{d}]^T \boldsymbol{R}^{-1}[\boldsymbol{z}-\boldsymbol{d}] + \\ &\frac{1}{2}\eta^T \boldsymbol{Q^{-1}} \eta \end{aligned} \qquad \text{(IV.11)}$$

Il faut minimiser cette dernière sous les contraintes d'égalité suivantes (avec $1 \le i \le n$):

$$z_i = \boldsymbol{H}_i \boldsymbol{M_{i-1}} \boldsymbol{M_{i-2}} \ldots \boldsymbol{M_0} \delta x_0 + H_i \sum_{k=1}^{i-1} (\boldsymbol{M_{i-1}} \boldsymbol{M_{i-2}} \ldots \boldsymbol{M_k} \eta_{k-1}) + \boldsymbol{H_i} \eta_{i-1} \qquad \text{(IV.12)}$$

Nous allons maintenant écrire sous forme matricielle les contraintes. Soit $\boldsymbol{K}$ la matrice définie par $n \times (n+1)$ blocs où chacun correspond à une matrice carrée $N \times N$:

$$\boldsymbol{K} = \begin{bmatrix} \boldsymbol{M}_0 & \boldsymbol{I} & & & & & & \\ \boldsymbol{M}_1 \boldsymbol{M}_0 & \boldsymbol{M}_1 & \boldsymbol{I} & & & & & \\ \ldots & \ldots & \ldots & & & & & \\ \boldsymbol{M}_{i-1} \ldots \boldsymbol{M}_0 & \boldsymbol{M}_{i-1} \ldots \boldsymbol{M}_1 & \boldsymbol{M}_{i-1} \ldots \boldsymbol{M}_2 & \ldots & \boldsymbol{M}_{i-1} & \boldsymbol{I} & & \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \\ \boldsymbol{M}_{n-1} \ldots \boldsymbol{M}_0 & \boldsymbol{M}_{n-1} \ldots \boldsymbol{M}_1 & \boldsymbol{M}_{n-1} \ldots \boldsymbol{M}_2 & \ldots & \ldots & \ldots & \boldsymbol{M}_{n-1} & \boldsymbol{I} \end{bmatrix}$$

On peut partitionner cette matrice en deux parties :

– la première est une matrice $(nN) \times N$ et correspond à la première colonne des blocs de la matrice ; elle sera notée $\boldsymbol{K_0}$

– la seconde correspond à la partie restante de $\boldsymbol{K}$: elle est de dimension $nN \times nN$; elle sera notée $\boldsymbol{K_1}$

$\boldsymbol{K}$ s'écrit donc en blocs sous la forme suivante : $\begin{bmatrix} \boldsymbol{K}_0 \boldsymbol{K}_1 \end{bmatrix}$

On note par $\boldsymbol{H}$ la matrice globale définie à partir des matrices jacobiennes des opérateurs d'observation H_i :

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{H}_1 & & & \\ & \boldsymbol{H}_2 & & \\ & & \ldots & \\ & & & \boldsymbol{H}_n \end{bmatrix}$$

$\boldsymbol{H}$ est de taille $p \times nN$ (car H_i est de taille $p_i \times N$).

Avec ces notations, l'ensemble des n contraintes s'exprime sous la forme matricielle suivante :

$$\boldsymbol{z} = \boldsymbol{H}\,\boldsymbol{K}\begin{bmatrix}\delta x_0 \\ \boldsymbol{\eta}\end{bmatrix} = \boldsymbol{H}\,\boldsymbol{K}_0 \delta x_0 + \boldsymbol{H}\,\boldsymbol{K}_1 \boldsymbol{\eta} \qquad \text{(IV.13)}$$

Ainsi, le problème peut s'exprimer par la minimisation de la fonction définie par la relation IV.11 sous la contrainte décrite par l'expression IV.13. Il s'agit donc d'un problème de programmation quadratique (les inconnues sont portées au carré dans l'expression IV.11), avec des contraintes d'égalité linéaires.

IV.2.2 Rappels de programmation quadratique sous contraintes linéaires d'égalités

Le problème précédent se ramène à la minimisation d'une fonction quadratique sous contraintes d'égalité :

$$min\, J(v) = \frac{1}{2} v^T A\, v - v^T f + c$$

Cette fonction doit être minimisée sous la contrainte suivante :

$$\boldsymbol{B^T} v = c$$

Avec les égalités suivantes, et $\boldsymbol{I}$ la matrice identité, nous aurons dans ce cas :

$$A = \begin{bmatrix} B_0^{-1} & 0 & 0 \\ 0 & \boldsymbol{Q^{-1}} & 0 \\ 0 & 0 & \boldsymbol{R^{-1}} \end{bmatrix}$$

$$v = \begin{pmatrix} \delta x_0 \\ \delta \eta \\ z \end{pmatrix}$$

$$B^T = [-\boldsymbol{H}\,\boldsymbol{K}, \boldsymbol{I}]$$

Ainsi, dans notre cas, A est inversible, B est de rang p, et $C = 0$. Si on considère la fonction suivante de Lagrange :

$$L(v) = J(v) + \lambda^T B^T v$$

La solution optimale correspond à

$$\frac{\delta L}{\delta v}=0 \Leftrightarrow A v+B^T \lambda=f \tag{IV.14}$$

$$\frac{\delta L}{\delta \lambda}=0 \Leftrightarrow B^T v=0 \tag{IV.15}$$

On obtient la formule suivante à partir de l'équation IV.14 :

$$v=A^{-1}(f-B\lambda) \tag{IV.16}$$

En remplaçant v par l'expression IV.16 dans l'équation IV.15 on obtient $B^T A^{-1}(f-B\lambda)=0$ qui est une équation simple en λ admettant une solution unique. Mais la résolution numérique dans notre cas est compliquée étant donné la nature complexe de la matrice B.

Si on remplace v par son expression en λ dans la fonction L, on obtient :

$$G(\lambda)=\frac{1}{2}[A^{-1}(f-B\lambda)]^T A[A^{-1}(f-B\lambda)]-[A^{-1}(f-B\lambda)]^T f + \lambda^T B^T[A^{-1}(f-B\lambda)]$$

Le développement de cette expression donne :

$$G(\lambda)=\frac{-1}{2}\lambda^T B^T A^{-1} B\lambda+ \lambda^T B^T A^{-1} f+ cte$$

Avec pour gradient :

$$\frac{\delta G}{\delta \lambda}=-B^T A^{-1} B\lambda+ B^T A^{-1} f=B^T A^{-1}(f-B\lambda)=B^T v$$

Ainsi pour résoudre l'équation IV.15, il suffit de trouver λ tel que $\frac{\delta G}{\delta \lambda}=0$.

Ceci peut être obtenu en cherchant le minimum de la fonction quadratique suivante, qui admet un minimum unique de part sa nature quadratique.

$$-G(\lambda)=\frac{1}{2}\lambda^T B^T A^{-1} B\lambda-\lambda^T B^T A^{-1} f$$

La minimisation peut se faire par la méthode itérative du gradient. On doit itérer jusqu'à l'obtention d'un gradient aussi petit que possible.

En résumé, le problème se ramène à minimiser $-G(\lambda)$ sans contrainte. Cette formulation est nommée la forme duale.

IV.2.3 Résolution du problème

IV.2.3.1 Passage en forme lagrangienne et expression en fonction de *m*

Pour rappel, le problème à résoudre est le suivant :

$$J(\delta x_{0,} \eta, z)=\frac{1}{2}\delta x_0^T B_0^{-1}\delta x_0+\frac{1}{2}[\boldsymbol{z}-\boldsymbol{d}]^T \boldsymbol{R}^{-1}[\boldsymbol{z}-\boldsymbol{d}]+\frac{1}{2}\eta^T \boldsymbol{Q}^{-1}\eta$$

Avec la contrainte :

$$\boldsymbol{z}=\boldsymbol{H}\cdot\boldsymbol{K}\begin{bmatrix}\delta x_0\\ \boldsymbol{\eta}\end{bmatrix}=\boldsymbol{H}\,\boldsymbol{K}_0\,\delta x_0+\boldsymbol{H}\,\boldsymbol{K}_1\boldsymbol{\eta}$$

Nous passons à la fonction de Lagrange correspondante, où $\boldsymbol{m}$ est le vecteur des coefficients de Lagrange. :

$$L(\delta x_{0,} \eta, \boldsymbol{z}; \boldsymbol{m})=J(\delta x, \eta, \boldsymbol{z})+ \boldsymbol{m}^T[z-\boldsymbol{H}\,\boldsymbol{K}\begin{bmatrix}\delta x\\ \eta\end{bmatrix}]$$

Nous avons donc :

$$\begin{aligned}L(\delta x_{0,}\boldsymbol{\eta}\,, \boldsymbol{z}\,; \boldsymbol{m})&=\frac{1}{2}\delta x_0^T \boldsymbol{B_0^{-1}}\delta x_0+\frac{1}{2}[\boldsymbol{z}-\boldsymbol{d}]^T \boldsymbol{R}^{-1}[\boldsymbol{z}-\boldsymbol{d}]+\frac{1}{2}\boldsymbol{\eta}^T \boldsymbol{Q^{-1}}\boldsymbol{\eta}\\ &+\boldsymbol{m^T}[z-\boldsymbol{H}\,\boldsymbol{K_0}\,\delta x_0-\boldsymbol{H}\,\boldsymbol{K_1}\boldsymbol{\eta}\,]\end{aligned} \qquad \text{(IV.17)}$$

Le vecteur m, de dimension $p=\sum_{i=1}^{n} p_i$ est formé par la concaténation de vecteurs m_i où chacun est de dimension p_i, et dont les composantes sont définies sur les p_i points de mesure au temps t_i, ainsi $\boldsymbol{m}^T=[\boldsymbol{m_1}^T, \boldsymbol{m_2}^T, \ldots, \boldsymbol{m_n}^T]$.

Nous allons écrire les conditions d'optimalité présentées au paragraphe IV.2.2.

Soit $\nabla_{\delta x_0} L=0$, et selon l'équation IV.17, et sachant que $\boldsymbol{B_0}$ est symétrique, on obtient :

$$\begin{aligned}&\delta x_0^T \boldsymbol{B_0^{-1}}-\boldsymbol{m}^T\boldsymbol{H}\,\boldsymbol{K_0}=0\\ &\delta x_0=\boldsymbol{B_0}\boldsymbol{K_0^T}\boldsymbol{H}^T\boldsymbol{m}\end{aligned}$$

On en déduit que δx_0 s'exprime en fonction de $\boldsymbol{m}$ par :

$$\delta x_0=\boldsymbol{B_0}\sum_{k=0}^{n-1}\boldsymbol{M_0^T}\boldsymbol{M_1^T}\ldots\boldsymbol{M_k^T}\boldsymbol{H_{k+1}^T}\boldsymbol{m_{k+1}} \qquad \text{(IV.18)}$$

La condition $\nabla_z L=0$ donne :

$$
\begin{aligned}
(\boldsymbol{z}-\boldsymbol{d})^T \boldsymbol{R}^{-1}+\boldsymbol{m}^T=0 \\
\boldsymbol{d}-\boldsymbol{z}=\boldsymbol{R}\,\boldsymbol{m} \\
\boldsymbol{m}=\boldsymbol{R}^{-1}(\boldsymbol{d}-\boldsymbol{z})
\end{aligned}
\qquad \text{(IV.19)}
$$

La condition $\nabla_{\eta} L=0$ donne :

$$\eta^T \boldsymbol{Q}^{-1}-\boldsymbol{m}^T \boldsymbol{H}\,\boldsymbol{K}_1=0$$

La matrice $\boldsymbol{Q}$ étant symétrique, on en déduit l'expression suivante:

$$\eta=\boldsymbol{Q}\,\boldsymbol{K}_1^T \boldsymbol{H}^T \boldsymbol{m} \qquad \text{(IV.20)}$$

IV.2.3.2 Passage en forme duale

En remplacant dans l'expression de L (équation IV.17) les variables $\delta x_0, z-d, \eta$ par leurs valeurs en fonction de $\boldsymbol{m}$ (selon les équations IV.18, IV.19 et IV.20), on obtient la fonction duale $G(m)$.

$$
\begin{aligned}
&\frac{1}{2}(\boldsymbol{B}_0 \boldsymbol{K}_0^T \boldsymbol{H}^T \boldsymbol{m})^T \boldsymbol{B}_0^{-1}(\boldsymbol{B}_0 \boldsymbol{K}_0^T \boldsymbol{H}^T \boldsymbol{m}) \\
&+\frac{1}{2}(\boldsymbol{R}\,\boldsymbol{m})^T \boldsymbol{R}^{-1}(\boldsymbol{R}\,\boldsymbol{m}) \\
&+\frac{1}{2}(\boldsymbol{Q}\,\boldsymbol{K}_1^T \boldsymbol{H}^T \boldsymbol{m})^T \boldsymbol{Q}^{-1}(\boldsymbol{Q}\,\boldsymbol{K}_1^T \boldsymbol{H}^T \boldsymbol{m}) \\
&+\boldsymbol{m}^T[\boldsymbol{d}-\boldsymbol{R}\,\boldsymbol{m}-\boldsymbol{H}\,\boldsymbol{K}_0(\boldsymbol{B}_0 \boldsymbol{K}_0^T \boldsymbol{H}^T \boldsymbol{m})-\boldsymbol{H}\,\boldsymbol{K}_1 \boldsymbol{Q}\,\boldsymbol{K}_1^T \boldsymbol{H}^T \boldsymbol{m}]
\end{aligned}
\qquad \text{(IV.21)}
$$

Après développement, nous obtenons :

$$
\begin{aligned}
&\frac{1}{2}\boldsymbol{m}^T(\boldsymbol{H}\,\boldsymbol{K}_0 \boldsymbol{B}_0 \boldsymbol{K}_0^T \boldsymbol{H}^T)\boldsymbol{m} \\
&+\frac{1}{2}\boldsymbol{m}^T \boldsymbol{R}\,\boldsymbol{m} \\
&+\frac{1}{2}\boldsymbol{m}^T(\boldsymbol{H}\,\boldsymbol{K}_1 \boldsymbol{Q}\,\boldsymbol{K}_1^T \boldsymbol{H}^T)\boldsymbol{m}-\boldsymbol{m}^T \boldsymbol{R}\,\boldsymbol{m}-\boldsymbol{m}^T(\boldsymbol{H}\,\boldsymbol{K}_0 \boldsymbol{B}_0 \boldsymbol{K}_0^T \boldsymbol{H}^T)\boldsymbol{m} \\
&-\boldsymbol{m}^T(\boldsymbol{H}\,\boldsymbol{K}_1 \boldsymbol{Q}\,\boldsymbol{K}_1^T \boldsymbol{H}^T)\boldsymbol{m}+\boldsymbol{m}^T d
\end{aligned}
$$

Soit :

$$-\frac{1}{2}\boldsymbol{m}^T[\,\boldsymbol{H}\,\boldsymbol{K}_0 \boldsymbol{B}_0 \boldsymbol{K}_0^T \boldsymbol{H}^T+\boldsymbol{R}+\boldsymbol{H}\,\boldsymbol{K}_1 \boldsymbol{Q}\,\boldsymbol{K}_1^T \boldsymbol{H}^T]\boldsymbol{m}+\boldsymbol{m}^T d$$

D'où la forme duale :

$$min\ -G(m)=\frac{1}{2}\boldsymbol{m}^T[\boldsymbol{H}\,\boldsymbol{K_0}\,\boldsymbol{B_0}\,\boldsymbol{K_0^T}\,\boldsymbol{H}^T+\boldsymbol{H}\,\boldsymbol{K_1}\,\boldsymbol{Q}\,\boldsymbol{K_1^T}\,\boldsymbol{H}^T+\boldsymbol{R}]\boldsymbol{m}-\boldsymbol{m}^T d \qquad \text{(IV.22)}$$

Sous sa forme duale, le problème est donc ramené au problème de minimisation de cette fonction quadratique sans contrainte.

IV.2.4 Premier algorithme

Nous avons exprimé l'ensemble des valeurs en fonction de $\boldsymbol{m}$, il nous faut maintenant exprimer les formules sous une forme utilisable par Yao. En effet, Yao lance les calculs sur l'ensemble des espace sur chaque pas de temps. Il nous faut donc des formules permettant de calculer des valeurs à partir du pas de temps précédent (pour une passe avant du modèle), ou à partir du pas de temps suivant (pour une rétropropagation).

De plus il nous faut içi minimiser la fonction $-G(m)$ définie dans la formule IV.22. Pour cela, nous allons utiliser le minimiseur M1QN3, qui est déjà utilisé pour l'algorithme à contrainte forte, et pour l'algorithme incrémental. Pour cela, nous aurons besoin des valeurs de la fonction $-G(m)$ et de son gradient relatif à m.

Nous devons donc içi exprimer les termes échelonnés dans le temps de la formule IV.22 sous la forme de fonctions au lieu de la forme matricielle.

IV.2.4.1 Expression du gradient de G(m)

A partir de l'équation IV.22, nous allons simplifier les éléments afin de retrouver une forme calculable par les algorithmes de Yao.

Au vu du fait que nous allons minimiser m en utilisant le minimiseur M1QN3/M2QN1, nous avons ici besoin également du gradient de $-G(m)$:

Nous allons reformuler l'équation $-G(m)$ sous une forme nous permettant de la dériver plus facilement :

$$-G(m)=\frac{1}{2}\boldsymbol{m}^T[\boldsymbol{H}\,\boldsymbol{K_0}\,\boldsymbol{B_0}\,\boldsymbol{K_0^T}\,\boldsymbol{H}^T+\boldsymbol{H}\,\boldsymbol{K_1}\,\boldsymbol{Q}\,\boldsymbol{K_1^T}\,\boldsymbol{H}^T+\boldsymbol{R}]\boldsymbol{m}-\boldsymbol{m}^T d$$

$$-G(m)=\frac{1}{2}\boldsymbol{m}^T[\boldsymbol{H}\,\boldsymbol{K_0}\,\boldsymbol{B_0}\,\boldsymbol{K_0^T}\,\boldsymbol{H}^T\boldsymbol{m}+\boldsymbol{H}\,\boldsymbol{K_1}\,\boldsymbol{Q}\,\boldsymbol{K_1^T}\,\boldsymbol{H}^T\boldsymbol{m}+\boldsymbol{R}\,\boldsymbol{m}]-\boldsymbol{m}^T d \qquad \text{(IV.23)}$$

$$-G(m)=\boldsymbol{m}^T(\frac{1}{2}[\boldsymbol{H}\,\boldsymbol{K_0}\,\boldsymbol{B_0}\,\boldsymbol{K_0^T}\,\boldsymbol{H}^T\boldsymbol{m}+\boldsymbol{H}\,\boldsymbol{K_1}\,\boldsymbol{Q}\,\boldsymbol{K_1^T}\,\boldsymbol{H}^T\boldsymbol{m}+\boldsymbol{R}\,\boldsymbol{m}]-d) \qquad \text{(IV.24)}$$

On peut donc calculer le gradient selon m de $-G(m)$:

$$\nabla_m - G(m) = \boldsymbol{H K_0 B_0 K_0^T H^T m + H K_1 Q K_1^T H^T m + R m} - d \qquad \text{(IV.25)}$$

D'autre part, en remplaçant le gradient IV.25 dans la formule IV.23, on obtient :

$$-G(m) = \frac{1}{2}\boldsymbol{m^T}[\nabla_m - G(m)] - \frac{1}{2}\boldsymbol{m^T} d$$

Cette formule montre que le calcul de $-G(m)$ peut se faire facilement en ayant calculé préalablement le gradient.

IV.2.4.2 Expression des opérations de rétropropagation

Observons maintenant les termes de -G(m). Nous allons reformuler les termes impliquant K_0, K_0^T, K_1, et K_1^T. En effet les opérations impliquées dans K_0 sont des passes avant en tangent linéaire du modèle, et pour K_0^T, il s'agit de rétropropagations d'adjoint. Pour le cas de K_1 et K_1^T, les différentes opérations sont cumulées. On définit les vecteurs suivants :

– $p(t_0)$ de dimension n tel que :

$$p(t_0) = \boldsymbol{K_0^T H^T m}$$

En développant la formule, on obtient :

$$p(t_0) = \sum_{k=0}^{n-1} M_0^T M_1^T \ldots M_k^T H_{k+1}^T m_{k+1}$$

Il s'agit de rétropropager toutes les composantes du vecteur m, qui sont définies sur l'ensemble des points de mesure. Le vecteur $p(t_0)$ correspond au résultat de cette rétropropagation sur l'espace de la grille au temps t_0.

On représente $\boldsymbol{K_1^T H^T m}$ par la concaténation de vecteurs $p(t_i)$ de taille N,avec $0 < i < n$. On a alors :

$$\boldsymbol{K_1^T H^T m} = \begin{pmatrix} p(t_1) \\ p(t_2) \\ \ldots \\ p(t_n) \end{pmatrix} = \boldsymbol{p} \qquad \text{(IV.26)}$$

$\boldsymbol{p}$ est ici le vecteur correspondant à l'ensemble des vecteurs $p(t_i)$. On peut en déduire :

$$p(t_i)=H_i^T m_i+\sum_{k=i}^{n-1} M_i^T M_{i+1}^T \dots M_k^T H_{k+1}^T m_{k+1}$$

En développant, on obtient :

$$p(t_i)=H_i^T m_i+M_i^T H_{i+1}^T m_{i+1}+\sum_{k=i+1}^{n-1} M_i^T M_{i+1}^T \dots M_k^T H_{k+1}^T m_{k+1}$$

$$p(t_i)=H_i^T m_i+M_i^T (H_{i+1}^T m_{i+1}+\sum_{k=i+1}^{n-1} M_{i+1}^T \dots M_k^T H_{k+1}^T m_{k+1})$$

Or on remarque que $H_{i+1}^T m_{i+1}+\sum_{k=i+1}^{n-1} M_{i+1}^T \dots M_k^T H_{k+1}^T m_{k+1}$ est l'expression de $p(t_{i+1})$. On peut donc en déduire pour $1 \le i \le n-1$, les relations suivantes :

$$p(t_n)=H_n^T m_n \quad \text{(IV.27)}$$

$$p(t_i)=H_i^T m_i+M_i^T p(t_{i+1}) \quad \text{(IV.28)}$$

$$p(t_0)=M_0^T (p(t_1)) \quad \text{(IV.29)}$$

Ces trois formules montrent que le calcul des vecteurs $p(t_i)$ se fait en initialisant tous les points de mesures en chaque pas de temps t_i par le vecteur m_i, et en rétro-propageant ces vecteurs m_i à travers le graphe, et en cumulant les calculs, de manière similaire à la rétropropagation de l'algorithme à contrainte forte.

Avec ces définitions on a $\nabla_m[-G(m)]=\boldsymbol{H K_0} B_0 p(t_0)+\boldsymbol{H K_1 Q p}+\boldsymbol{R m}-\boldsymbol{d}$

IV.2.4.3 Expression des opérations de tangent linéaire

Nous allons maintenant exprimer les opérations $\boldsymbol{K_0} B_0 p(t_0)$ et $\boldsymbol{K_1 Q p}$ qui interviennent dans le calcul de $\nabla_m - G(m)$.

On sait que :

$$\delta x_0 = B_0 \boldsymbol{K_0^T H^T m} = B_0 p(t_0) \quad \text{(IV.30)}$$

D'autre part, on a :

$$\boldsymbol{Q K_1^T H^T m}=\boldsymbol{Q}\begin{bmatrix} p(t_1) \\ \dots \\ p(t_n) \end{bmatrix}=\begin{bmatrix} \boldsymbol{Q_1} p(t_1) \\ \dots \\ \boldsymbol{Q_n} p(t_n) \end{bmatrix}$$

On définit le vecteur $\delta \boldsymbol{x}$ de dimension $n \times N$ par :

$$\delta \boldsymbol{x} = \boldsymbol{K_1 Q K_1^T H^T m} = \begin{pmatrix} \delta x_1 \\ \delta x_2 \\ \dots \\ \delta x_n \end{pmatrix} = \boldsymbol{K_1 Q} \begin{pmatrix} p(t_1) \\ p(t_2) \\ \dots \\ p(t_n) \end{pmatrix} = \boldsymbol{K_1} \begin{pmatrix} \boldsymbol{Q}\, p(t_1) \\ \boldsymbol{Q}\, p(t_2) \\ \dots \\ \boldsymbol{Q}\, p(t_n) \end{pmatrix} \quad \text{(IV.31)}$$

Où chaque δx_i ($1 \le i \le n$) est un vecteur de dimension N .

De par la structure de K_1 , on peut déduire que, pour $2 \le i \le n$:

$$\delta x_i = \boldsymbol{M_{i-1}} \dots \boldsymbol{M_1} Q_1\, p(t_1) + \boldsymbol{M_{i-1}} \dots \boldsymbol{M_2} Q_2\, p(t_2) + \dots + \boldsymbol{M_{i-1}} Q_{i-1}\, p(t_{i-1}) + Q_i\, p(t_i)$$
$$\delta x_i = \boldsymbol{M_{i-1}} [\, \boldsymbol{M_{i-2}} \dots \boldsymbol{M_1} Q_1\, p(t_1) + \boldsymbol{M_{i-2}} .. \boldsymbol{M_2} Q_2\, p(t_2) + .. + Q_{i-1}\, p(t_{i-1})] + Q_i\, p(t_i)$$

On remarque que $\boldsymbol{M_{i-2}} \dots \boldsymbol{M_1} Q_1\, p(t_1) + \boldsymbol{M_{i-2}} \dots \boldsymbol{M_2} Q_2\, p(t_2) + \dots + Q_{i-1}\, p(t_{i-1})$ est l'expression de δx_{i-1} .

On peut calculer donc les différentes composantes du vecteur $\delta \boldsymbol{x}$ en appliquant les expressions suivantes :

$$\delta x_1 = Q_1\, p(t_1) \quad \text{(IV.32)}$$

$$\delta x_i = \boldsymbol{M_{i-1}} \delta x_{i-1} + Q_i\, p(t_i) \quad \text{(IV.33)}$$

En résumé, on peut obtenir les $p(t_i)$ en rétro-propageant les m à partir de l'ensemble des points de mesure, et obtenir $\delta \boldsymbol{x}$ en effectuant une passe avant de tangent linéaire à partir de δx_1 en récupérant lors de la passe avant les valeurs $Q_i\, p(t_i)$ précédemment calculées.

On applique les formules obtenues aux équations IV.26 et IV.31 aux formules IV.24 et IV.25, en prenant $\boldsymbol{\delta x}$ comme vecteur par blocs formé de δx_1 en première position et de δx_i dans les suivantes. On obtient :

$$-G(m) = \frac{1}{2} \boldsymbol{m^T} [\, \boldsymbol{H K_0}\, \delta x_0 + \boldsymbol{H}\, \delta \boldsymbol{x} + \boldsymbol{R}\,] \boldsymbol{m} - \boldsymbol{m^T} d \quad \text{(IV.34)}$$

$$\nabla_m - G(m) = \boldsymbol{H K_0}\, \delta x_0 + \boldsymbol{H}\, \delta \boldsymbol{x} + \boldsymbol{R m} - d \quad \text{(IV.35)}$$

$\boldsymbol{K_0}\, \delta x_0$ *représente la propagation des tangents linéaires à partir de δx_0 , en récupérant les résultats pour chaque t_i .*

Rappelons l'expression de $-G(m)$ en fonction de $\nabla_m - G(m)$:

$$-G(m)=\frac{1}{2}m^T[\nabla_m - G(m)]-\frac{1}{2}m^T d \ .$$

IV.2.4.4 Expression de η

On sait que, selon le tableau IV.1 :

$$\boldsymbol{\eta = Q K_1^T H^T m}$$

En tenant compte de la formule IV.26, on en déduit que :

$$\eta_i = Q_i p(t_i) \qquad \text{(IV.36)}$$

IV.2.5 Rappel des variables en présence

Pour rappel, voici un tableau synthétisant l'ensemble des variables utilisées, ce pour une meilleure compréhension des algorithmes.

Variable	Nom	Explication	Forme itérative	Forme matricielle
x_i	Variable d'état du modèle au temps t_i	Calculée par le modèle sans bruit et en partant de $x_0 = x_b$	$x_i = M_{i-1}(x_{i-1})$	
x_i^t	Variable d'états, estimation au temps t	Calculées en gardant le bruit	$x_i^t = M_{i-1}(x_{i-1}^t) + \eta_i$	
m_i	Coefficients de Lagrange	Coefficients des contraintes du Lagrangien	$\boldsymbol{R^{-1}}(y^{obs} - H\,M\,x^b - \boldsymbol{z})$	$\boldsymbol{R^{-1}(d-z)}$
d_i	Vecteur d'innovation	Différence entre passe avant du modèle sur ébauche et observation	$y_i^{obs} - H_i M_{i-1} \circ M_{i-2} \circ \ldots \circ M_0(x^b)$	-
z_i		Passe avant en tangent linéaire cumulée à partir de δx et $\boldsymbol{\eta}$	$\boldsymbol{H_i M_{i-1} M_{i-2} \ldots M_0 \delta x_0} + \sum_{k=1}^{i-1} \boldsymbol{H_i M_{i-1} M_{i-2} \ldots M_k \eta_{k-1}} + \boldsymbol{H_i \eta_{i-1}}$	$\boldsymbol{H K_0 \delta x_0 + H K_1 \eta}$
η_i	Erreur sur modèle		$x_i^t = M_{i-1}(x_{i-1}^t) + \eta_{i-1}$	$\boldsymbol{Q K_1^T H^T m}$
$p(t_n)$		Résultat de la rétroprogation des coefficients de Lagrange.	$\boldsymbol{H_n^T m_n}$	
$p(t_i)$			$\boldsymbol{H_i^T m_i + M_i^T p(t_{i+1})}$	$\boldsymbol{K_1^T H^T m}$
$p(t_0)$			$\boldsymbol{M_0^T p(t_1)}$	$\boldsymbol{K_0^T H^T m}$
δx_0		Décalage de x_0^t attendu.	$B_0 p(t_0)$	$B_0 \boldsymbol{K_0^T H^T m}$
δx_1		Tangent linéaire cumulé de $p(t_i)$	$Q_1 p(t_1)$	
δx_i			$\boldsymbol{M_{i-1}} \delta x_{i-1} + Q_i p(t_i)$	
$-G(m)$	Valeur de la fonction coût	Scalaire	$\frac{1}{2}\boldsymbol{m^T}[H_0 \boldsymbol{K_0} + \boldsymbol{H \delta_x} + \boldsymbol{R m}] - \boldsymbol{m^T}(\boldsymbol{d} - \boldsymbol{H K_1 \eta})$	
$-\nabla G(m)$	Gradient de la fonction coût	Défini sur ensemble des observations	$[H_0 \boldsymbol{K_0} + \boldsymbol{H \delta_x} + \boldsymbol{R m}] - (\boldsymbol{d} - \boldsymbol{H K_1 \eta})$	

Tableau IV.1: Synthèse des variables en présence

IV.2.5.1 Algorithme

Avec les éléments décrits auparavant, ainsi que les fonctionnalités actuelles de Yao, il est possible de prévoir un algorithme général pour le calcul . En effet, $p(t_i)$ et δx_i

sont tous deux calculables par les algorithmes de propagation et de rétropropagation sur le graphe modulaire de Yao.

Voici une première proposition d'algorithme pour le 4Dvar à contrainte faible :

1. Calcul d'une passe avant du modèle (pour obtenir les valeurs $HM(\boldsymbol{x}^b)$ en chaque point de maille) en prenant comme valeur initiale x^b, afin de calculer $\boldsymbol{d}$ (tableau IV.1).

2. On initialise η_i à zéro pour chaque pas de temps t_i. On prend δx_0 initialisé également à zéro. A partir de ces valeurs, on calcule une passe avant du tangent linéaire afin d'initialiser chaque m_i par $\boldsymbol{R^{-1}}(d_i - H_i M_{i-1} M_{i-2} \ldots M_0 \delta x_0)$, ce pour chaque observation.

3. Calcul des vecteurs $p(t_i)$ à l'aide des équations IV.28 et IV.29, en partant de l'équation IV.27. On initialise tous les points de mesures pour chaque pas de temps par m_i, puis on rétropropage en ajoutant le terme $H_i^T m_i$ pour chaque pas de temps et pour chaque point de maille. Ceci permet de calculer les vecteurs $p(t_i)$. Nous gardons ces valeurs en mémoire pour les calculs futurs.

4. Calcul de δx_0 par $B_0 p(t_0)$ (expression IV.30), puis de δx_1 via $Q_1 p(t_1)$ (expression IV.32).

5. Passe avant du tangent linéaire à partir de δx_1 initialisé au temps t_1, en ajoutant le terme $Q_i p(t_i)$ pour chaque pas de temps et chaque point de maille (voir équation IV.33). On obtient ainsi l'ensemble des vecteurs δx_i, que l'on stocke.

6. Calcul de $-G(m)$ et $\nabla_m - G(m)$ à partir des équations IV.34 et IV.35.

7. Effectuer une minimisation de $-G(m)$ afin de calculer une nouvelle valeur de m.

8. Si les critères d'arrêt de la minimisation ne sont pas atteints, retour en phase 3.

A la fin de l'algorithme, on obtient les différentes erreurs modèle η_i, en appliquant le calcul de la phase 3 (grâce à l'équation IV.36). Le vecteur δx_0 calculé en phase 4 nous permet de trouver la nouvelle valeur des paramètres $x_0 = x^b + \delta x_0$.

Cependant il faut garder à l'esprit que nous nous appuyons depuis le départ sur une approximation de la fonction coût initiale en procédant à une linéarisation locale de celle-ci. L'algorithme minimise alors cette approximation qui est une fonction quadratique. Nous allons donc introduire dans la partie suivante une boucle extérieure supplémentaire,

nous permettant de refaire une nouvelle approximation de la fonction coût autour de la nouvelle valeur x_0 calculée. L'algorithme général résultant pourra s'apparenter ainsi dans l'esprit à l'algorithme incrémental décrit dans la partie I.2.1.4.

IV.2.6 Boucle externe et algorithme final

Nous avons donc besoin de nous replacer sur la fonction coût après un certain nombre d'itérations de la boucle interne. De plus, la passe avant (phase 1 dans la partie précédente) ne tient pas compte des erreurs modèle calculées. En effet, le vecteur d'innovation n'est pas modifié. Nous allons donc incorporer au niveau de la passe avant les erreurs du modèle η_i. Nous allons donc proposer içi une version améliorée de l'algorithme précédent, qui représentera la boucle interne. La boucle externe consistera à se replacer sur la fonction coût réelle, après les itérations de la boucle interne.

Retenons pour la suite les notations suivantes :

- x_0^t : l'estimation de l'état du vecteur de contrôle au début d'une boucle externe. Il sera initialisé avec le vecteur x^b au début de la première boucle externe. En génral il correspond à l'estimation calculée à la fin de l'itération externe précédente ;
- $x_0^{t'}$: l'estimation de l'état « réel » calculé à l'itération courante de la boucle interne. Il correspond à $x_0^t + \delta x_0$ où δx_0 représente l'estimation à l'itération interne courante ;
- x_i^t : l'estimation de l'état « réel » au temps i calculé avec l'équation IV.1 à la fin de l'itération externe précédente ;
- On peut retrouver $x_i^{t'}$ en calculant une passe avant du modèle à partir de $x_0^{t'}$ et en y ajoutant les erreurs du modèle η_i calculées à l'itération précédente de la boucle externe, ainsi que leur accroissements $\delta\eta_i$ calculés à l'itération interne courante. $\delta\eta_i$ est initialisé à zéro pour chaque itération interne. η_i correspond ainsi au cumul des bruits calculés par l'ensemble des itérations externes et internes précédentes.

On peut donc considérer les égalités suivantes :

$$x_1^{t'} = M_0\left(x_0^t + \delta x_0\right) + \eta_0 + \delta\eta_0$$
$$x_2^{t'} = M_1\left(x_1^{t'}\right) + \eta_1 + \delta\eta_1$$

$$x_i^{t'} = M_{i-1}(x_{i-1}^{t'}) + \eta_{i-1} + \delta\eta_{i-1} \tag{IV.37}$$

Il s'agit dans ce cas de minimiser :

$$\begin{aligned} J(\delta x_0, \delta\eta) = & \frac{1}{2}(x_0^t - x^b + \delta x_0)^T B_0^{-1}(x_0^t - x^b + \delta x_0) \\ & + \frac{1}{2}\sum_{i=1}^{n}[H(x_i^{t'}) - y_i^{obs}]^T R_i^{-1}[H(x_i^{t'}) - y_i^{obs}] \\ & + \frac{1}{2}\sum_{i=1}^{n}(\eta_{i-1} + \delta\eta_{i-1})^T Q_i^{-1}(\eta_{i-1} + \delta\eta_{i-1}) \end{aligned} \tag{IV.38}$$

Selon le même principe que dans la partie IV.2.1.2, nous allons reformuler l'expression IV.38 en procédant à des linéarisations locales de $x_i^{t'} = M_{i-1}(x_{i-1}^{t'}) + \eta_{i-1}$.

Ainsi pour $i = 1$ on a :

$$\begin{aligned} x_1^{t'} &= M(x_0^t + \delta x_0) + \eta_0 + \delta\eta_0 \\ x_1^{t'} &\simeq M(x_0^t) + \boldsymbol{M_0}\delta x_0 + \eta_0 + \delta\eta_0 \\ x_1^{t'} &\simeq (M(x_0^t) + \eta_0) + \boldsymbol{M_0}\delta x_0 + \delta\eta_0 = x_1^t + \boldsymbol{M_0}\delta x_0^t + \delta\eta_0 \end{aligned}$$

$\boldsymbol{M_0}$ étant la matrice jacobienne du modèle M_0 calculé à l'état x_0^t.

En généralisant ce résultat jusqu'au pas de temps t_i, on obtient :

$$x_i^{t'} \simeq x_i^t + \boldsymbol{M_{i-1}M_{i-2}}..\boldsymbol{M_0}\delta x_0 + \sum_{k=1}^{i-1}[\boldsymbol{M_{i-1}}...\boldsymbol{M_k}\delta\eta_{k-1}] + \delta\eta_{i-1} \tag{IV.39}$$

Où $\boldsymbol{M_j}$ correspond à la matrice jacobienne du modèle M calculé à l'état $x_{j-1}^{t'}$

On applique l'opérateur d'observation H à l'expression IV.39, puis on le linéarise localement. On obtient, avec H_i calculé au point x_i^t :

$$H(x_i^{t'}) \simeq H(x_i^t) + \boldsymbol{H_i M_{i-1}}..\boldsymbol{M_0}\delta x_0 + \boldsymbol{H_i}\sum_{k=1}^{i-1}[\boldsymbol{M_{i-1}}..\boldsymbol{M_k}\delta\eta_{k-1}] + \boldsymbol{H_i}\delta\eta_{i-1} \tag{IV.40}$$

On définit d_i et z_i par :

$$z_i = \boldsymbol{H_i M_{i-1}}..\boldsymbol{M_0}\delta x_0 + \boldsymbol{H_i}\sum_{k=1}^{i-1}[\boldsymbol{M_{i-1}}..\boldsymbol{M_k}\delta\eta_{k-1}] + \boldsymbol{H_i}\delta\eta_{i-1} \tag{IV.41}$$

$$d_i = y_i^{obs} - H_i(x_i^t) = y_i^{obs} - H_i(x_i^{t'}) + z_i \tag{IV.42}$$

On a alors $H(x_i^{t'}) - y_i^{obs} \simeq z_i - d_i$. La réécriture de $\tilde{J}$ donne :

$$J(\delta x_{0,}\delta\eta,\boldsymbol{z})=\frac{1}{2}(x_0^t-x^b+\delta x_0)^T B_0^{-1}(x_0^t-x^b+\delta x_0)+\frac{1}{2}\sum_{i=1}^{n}(z_i-d_i)^T R_i^{-1}(z_i-d_i)$$
$$+\frac{1}{2}\sum_{i=1}^{n}(\eta_{i-1}+\delta\eta_{i-1})^T Q_i^{-1}(\eta_{i-1}+\delta\eta_{i-1}) \qquad \text{(IV.43)}$$

On considère la matrice $\boldsymbol{K}$ de dimensions $nN\times(n+1)N$ et définie par blocs comme suit :

$$\boldsymbol{K}=\begin{bmatrix} \boldsymbol{M}_0 & \boldsymbol{I} & & & & & & \\ \boldsymbol{M}_1\boldsymbol{M}_0 & \boldsymbol{M}_1 & \boldsymbol{I} & & & & & \\ \cdots & \cdots & \cdots & & & & & \\ \boldsymbol{M}_{i-1}\ldots\boldsymbol{M}_0 & \boldsymbol{M}_{i-1}\ldots\boldsymbol{M}_1 & \boldsymbol{M}_{i-1}\ldots\boldsymbol{M}_2 & \ldots & \boldsymbol{M}_{i-1} & \boldsymbol{I} & & \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\ \boldsymbol{M}_{n-1}\ldots\boldsymbol{M}_0 & \boldsymbol{M}_{n-1}\ldots\boldsymbol{M}_1 & \boldsymbol{M}_{n-1}\ldots\boldsymbol{M}_2 & \ldots & \ldots & \ldots & \boldsymbol{M}_{n-1} & \boldsymbol{I} \end{bmatrix} \qquad \text{(IV.44)}$$

En notant par $\boldsymbol{K_0}$ la première colonne des blocs de $\boldsymbol{K}$ et par $\boldsymbol{K_1}$ les n colonnes des blocs restants, on peut écrire alors :

$$\boldsymbol{K}=[\boldsymbol{K_{0,}}\boldsymbol{K_1}] \qquad \text{(IV.45)}$$

Ainsi, le problème peut se ramener à la minimisation de la fonction quadratique définie par IV.43 sous les contraintes linéaires :

$$\boldsymbol{z}=\boldsymbol{H}\,\boldsymbol{K}\begin{bmatrix}\delta x_0\\ \delta\eta\end{bmatrix}=\boldsymbol{H}\,\boldsymbol{K_0}\,\delta x_0+\boldsymbol{H}\,\boldsymbol{K_1}\,\delta\eta \qquad \text{(IV.46)}$$

La résolution nous amène à introduire le lagrangien de ce problème, où $\boldsymbol{m}$ est le vecteur composé par les coefficients de Lagrange :

$$\begin{aligned} &L(\delta x_{0,}(\delta\eta),\boldsymbol{z},\boldsymbol{m})=\\ &\frac{1}{2}\delta x_0^T B_0^{-1}\delta x_0+(x_0^t-x_0^b)^T B_0^{-1}\delta x_0\\ &+\frac{1}{2}(z_i-d_i)^T R_i^{-1}(z_i-d_i)\\ &+(\frac{1}{2}\eta_{i-1}+\delta\eta_{i-1}{}^T)Q_i^{-1}(\eta_{i-1}+\delta\eta_{i-1})\\ &+\boldsymbol{m}^T[\boldsymbol{z}-\boldsymbol{H}\,\boldsymbol{K_0}\,\delta x_0-\boldsymbol{H}\,\boldsymbol{K_1}\,\delta\eta] \end{aligned} \qquad \text{(IV.47)}$$

On calcule les points où le gradient de chaque inconnue s'annule :

– $\nabla_{\delta x_0}L=0$

$$\delta x_0^T B_0^{-1} + (x_0^t - X_0^b)^T B_0^{-1} - \boldsymbol{m}^T \boldsymbol{H}\, \boldsymbol{K_0} = 0$$
$$\delta x_0^T B_0^{-1} = \boldsymbol{m}^T \boldsymbol{H}\, \boldsymbol{K_0} - (x_0^t - x^b)^T B_0^{-1}$$
$$\delta x_0^T = \boldsymbol{m}^T \boldsymbol{H}\, \boldsymbol{K_0} B_0 - (x_0^t - x^b)^T$$
$$\delta x_0 = B_0 \boldsymbol{K_0^T} \boldsymbol{H}^T \boldsymbol{m} - (x_0^t - x^b) \qquad \text{(IV.48)}$$

- $\nabla_z L = 0$

$$(\boldsymbol{z} - \boldsymbol{d})^T \boldsymbol{R^{-1}} + \boldsymbol{m}^T = 0$$
$$\boldsymbol{m} = \boldsymbol{R^{-1}}(\boldsymbol{d} - \boldsymbol{z}) \qquad \text{(IV.49)}$$
$$\boldsymbol{R}\,\boldsymbol{m} = \boldsymbol{d} - \boldsymbol{z} \qquad \text{(IV.50)}$$

- $\nabla_{\delta\eta} L = 0$

$$\boldsymbol{\delta\eta}^T \boldsymbol{Q^{-1}} + \boldsymbol{\eta}^T \boldsymbol{Q^{-1}} - \boldsymbol{m}^T \boldsymbol{H}\, \boldsymbol{K_1} = 0$$
$$\boldsymbol{\delta\eta}^T = -\boldsymbol{\eta}^T + \boldsymbol{m}^T \boldsymbol{H}\, \boldsymbol{K_1} \boldsymbol{Q}$$
$$\delta\eta = -\eta + \boldsymbol{Q K_1^T H}^T \boldsymbol{m} \qquad \text{(IV.51)}$$

Dans l'expression IV.51, il est à noter qu'il est possible d'exprimer $\boldsymbol{\delta\eta}$ en fonction de $\boldsymbol{\eta}$, nous pouvons donc retirer cette variable de la fonction coût.

On exprime l'expression de la nouvelle fonction coût en fonction de $\boldsymbol{m}$, afin de passer à la forme duale. On a donc :

$$G(m) = \frac{1}{2}[B_0 \boldsymbol{K_0^T} \boldsymbol{H}^T \boldsymbol{m} - (x_0^t - x^b)]^T B_0^{-1} [B_0 \boldsymbol{K_0^T} \boldsymbol{H}^T \boldsymbol{m} - (x_0^t - x^b)]$$
$$+ \frac{1}{2}(\boldsymbol{z} - \boldsymbol{d})^T \boldsymbol{R^{-1}} (\boldsymbol{z} - \boldsymbol{d}) + \frac{1}{2}[-\boldsymbol{\eta}^T + \boldsymbol{m}\, \boldsymbol{H}\, \boldsymbol{K_1} \boldsymbol{Q}] \boldsymbol{Q^{-1}} [-\boldsymbol{\eta} + \boldsymbol{Q K_1^T H}^T \boldsymbol{m}]$$
$$+ \boldsymbol{\eta}^T \boldsymbol{Q^{-1}} [-\boldsymbol{\eta} + \boldsymbol{Q K_1^T H}^T \boldsymbol{m}]$$
$$+ \boldsymbol{m}^T [\boldsymbol{d} - \boldsymbol{R}\,\boldsymbol{m} - \boldsymbol{H}\, \boldsymbol{K_0} B_0 \boldsymbol{K_0^T} \boldsymbol{H}^T \boldsymbol{m} + \boldsymbol{H}\, \boldsymbol{K_0} (x_0^t - x^b) + \boldsymbol{H}\, \boldsymbol{K_1} \boldsymbol{\eta} - \boldsymbol{H}\, \boldsymbol{K_1} \boldsymbol{Q K_1^T H}^T \boldsymbol{m}]$$

On développe $G(m)$ et on la réorganise en deux expressions: IV.52 et IV.53.

$$-G(m) = \frac{1}{2}(B_0 \boldsymbol{K_0^T} \boldsymbol{H}^T \boldsymbol{m})^T B_0^{-1} (B_0 \boldsymbol{K_0^T} \boldsymbol{H}^T \boldsymbol{m}) + \frac{1}{2}(\boldsymbol{R}\,\boldsymbol{m}) \boldsymbol{R^{-1}} (\boldsymbol{R}\,\boldsymbol{m})$$
$$+ \frac{1}{2}[\boldsymbol{m}^T \boldsymbol{H}\, \boldsymbol{K_1} \boldsymbol{Q}] \boldsymbol{Q^{-1}} [\boldsymbol{Q K_1^T H}^T \boldsymbol{m}] \qquad \text{(IV.52)}$$
$$+ \boldsymbol{m}^T [\boldsymbol{d} - \boldsymbol{R}\,\boldsymbol{m} - \boldsymbol{H}\, \boldsymbol{K_0} B_0 \boldsymbol{K_0^T} \boldsymbol{H}^T \boldsymbol{m} - \boldsymbol{H}\, \boldsymbol{K_1} \boldsymbol{Q}^T \boldsymbol{K_1^T} \boldsymbol{H}^T \boldsymbol{m}]$$
$$+ \frac{1}{2}(x_0^t - x^b) B_0^{-1} (x_0^t - x^b) - (x_0^t - x^b)^T B_0^{-1} [B_0 \boldsymbol{K_0^T} \boldsymbol{H}^T \boldsymbol{m}]$$
$$+ \frac{1}{2}\boldsymbol{\eta}^T \boldsymbol{Q^{-1}} \boldsymbol{\eta} - \boldsymbol{\eta}^T \boldsymbol{Q^{-1}} (\boldsymbol{Q K_1^T H}^T \boldsymbol{m}) - \boldsymbol{\eta}^T \boldsymbol{Q^{-1}} \boldsymbol{\eta} + \eta^T \boldsymbol{Q^{-1}} (\boldsymbol{Q K_1^T H}^T \boldsymbol{m}) \qquad \text{(IV.53)}$$
$$+ \boldsymbol{m}^T [\boldsymbol{H}\, \boldsymbol{K_0} (x_0^t - x^b) + \boldsymbol{H}\, \boldsymbol{K_1} \boldsymbol{\eta}]$$

On simplifie l'expression IV.53 et on en retire les expressions constantes par rapport à $\boldsymbol{m}$. Il reste :

$$\boldsymbol{m}^T \boldsymbol{H}\boldsymbol{K}_1\boldsymbol{\eta}$$

On peut donc en déduire que la nouvelle fonction $G(m)$:

$$G(m)=-\frac{1}{2}\boldsymbol{m}^T[\boldsymbol{H}\boldsymbol{K}_0 B_0 \boldsymbol{K}_0^T \boldsymbol{H}^T+\boldsymbol{H}\boldsymbol{K}_1\boldsymbol{Q}\boldsymbol{K}_1^T\boldsymbol{H}^T+\boldsymbol{R}]\boldsymbol{m}+\boldsymbol{m}^T\boldsymbol{d}+\boldsymbol{m}^T\boldsymbol{H}\boldsymbol{K}_1\boldsymbol{\eta}$$

Après le regroupement des deux derniers termes, on obtient l'expression suivante, avec $\boldsymbol{d}'=\boldsymbol{d}-\boldsymbol{H}\boldsymbol{K}_1\boldsymbol{\eta}$:

$$G(m)=-\frac{1}{2}\boldsymbol{m}^T[\boldsymbol{H}\boldsymbol{K}_0 B_0 \boldsymbol{K}_0^T \boldsymbol{H}^T+\boldsymbol{H}\boldsymbol{K}_1\boldsymbol{Q}\boldsymbol{K}_1^T\boldsymbol{H}^T+\boldsymbol{R}]\boldsymbol{m}+\boldsymbol{m}^T\boldsymbol{d}' \qquad \text{(IV.54)}$$

Ainsi, la forme duale du problème de minimisation de la fonction quadratique IV.43 sous les contraintes linéaires IV.46 se réduit à $max\ G(m)$, et les variables de la forme primale obtenues en appliquant les formules IV.48, IV.50, et IV.51.

Par la suite on minimise la fonction $-G(m)$. Les vecteurs $p_i(x)$ et δx_i définies dans les parties précédentes peuvent être introduits dans ce cas. A partir des formules IV.34 et IV.35, on peut donc énoncer :

$$G(m)=\frac{-1}{2}\boldsymbol{m}^T[\boldsymbol{H}\boldsymbol{K}_0\delta x_0+\boldsymbol{H}\boldsymbol{\delta}\boldsymbol{x}+\boldsymbol{R}\boldsymbol{m}]+\boldsymbol{m}^T\boldsymbol{d}' \qquad \text{(IV.55)}$$

Et le gradient selon $\boldsymbol{m}$:

$$\nabla_m -G(m)=\boldsymbol{H}\boldsymbol{K}_0\delta x_0+\boldsymbol{H}\boldsymbol{\delta}\boldsymbol{x}+\boldsymbol{R}\boldsymbol{m}-\boldsymbol{d}' \qquad \text{(IV.56)}$$

On remarque que l'expression de $-G(m)$ est analogue à celle de IV.23 définie dans la partie précédente. On peut alors utiliser la même méthode pour calculer $-G(m)$ et son gradient.

IV.2.7 Rappel des variables en présence

Les variables définies au tableau IV.1 correspondent à celles utilisées lors de la première itération externe. En effet, celle-ci est initialisée avec $\eta=0$, et $x_0^t=x^b$. Par contre les matrices jacobiennes $\boldsymbol{M}_i$ et $\boldsymbol{H}_i$ seront calculées aux points suivants :

- au début d'une itération externe, la variable d'état x_o^t est initialisée à la valeur de la variable x_0^t calculée à la fin de l'itération externe précédente. On a donc :
  - x_0^t (à l'itération k) = x_0^t (à l'itération $k-1$) + δx (modification de l'itération $k-1$)

- la variable d'état au temps t_i est définie par $x_i^t = M_{i-1}(x_{i-1}^t) + \eta_{i-1}$, η_i étant l'erreur modèle calculée à la fin de l'itération externe précédente.
- Les matrices jacobiennes $\boldsymbol{M_i}$ sont alors calculées aux états x_i^t.
- $d_i = y_i^{obs} - H_i(x_i^t)$
- $\boldsymbol{d'} = \boldsymbol{d} - \boldsymbol{H K_1} \eta$, $\boldsymbol{d}$ étant le vecteur de composantes d_i et η le vecteur de composantes η_i qui sont calculées à la fin de l'itération externe précédente.

Avec ces modifications, les autres variables s'expriment comme celles du tableau, mais en remplaçant $\boldsymbol{d}$ par $\boldsymbol{d'}$.

IV.2.8 Algorithme

Voici le nouvel algorithme correspondant aux calculs précédemment décrits :

1. Initialisation de η à 0, δx_0 à 0 et x_0^t à x^b.
2. Boucle externe

   On initialise $\delta\eta = 0$

   1. Calcul des x_i^t et des d_i : passe avant (*FORWARD*) avec cumul de l'erreur modèle $\boldsymbol{\eta}$ à partir de x_0^t (formule IV.42, dans le but de calculer $\boldsymbol{d}$). Les valeurs à garder sont sur l'espace des observations. Calcul de $\boldsymbol{d}$.
   2. Calcul de $\boldsymbol{H K_1} \eta$: tangent linéaire (*LINWARD*) avec cumul de $\boldsymbol{\eta}$, puis calcul de $\boldsymbol{d'} = \boldsymbol{d} - \boldsymbol{H K_1} \eta$
   3. Boucle interne

      1. Adjoint (*BACKWARD*) en rétropropageant les variables $\boldsymbol{m}$ définies aux points de mesures, dans le but de calculer δx_0 et les $p(t_i)$
      2. Tangent linéaire (*LINWARD*) à partir de δx_1 avec cumul de $Q_i p(t_i)$ dans le but de calculer $\boldsymbol{\delta} x$.
      3. Tangent linéaire (*LINWARD*) à partir de δx_0 dans le but de calculer $\boldsymbol{H K_0} \delta x_0$
      4. Pour chaque observation, calcul de $-G(m_i)$, et de $-\nabla_m G(m_i)$.
      5. Minimisation à l'aide des valeurs calculée, récupération nouveau m_i.

4. Suite boucle externe : ajout de δx_0 à x_0^t pour obtenir la nouvelle valeur x_0^t. Calcul du nouveau vecteur η = précédent $\eta + \delta\eta = \boldsymbol{Q}\,\boldsymbol{K}_1^T\,\boldsymbol{H}^T\boldsymbol{m}$. Ces deux vecteurs initialisent l'itération externe suivante.

IV.2.9 Routines à implémenter

Par rapport aux fonctionnalités existantes et variables de Yao, les routines suivantes ont été implémentées :

A) passe avant cumulant les erreurs modèle, partant d'un x_b donné

B) tangent linéaire cumulant les erreurs modèle, partant d'un η donné

C) adjoint pour calculer $p(t_i)$ en se servant des variables m

D) tangent linéaire cumulant les $Q_i\,p(t_i)$, partant de δx_1

E) tangent linéaire normal, partant de δx_0

F) calcul de la fonction coût spécifique au dual et de son gradient

G) minimisation dans l'ensemble des observations

L'algorithme commence par une initialisation de x_0 à l'ébauche. Quand pour les autres algorithmes, on peut indiquer un « first guess » différent de l'ébauche, pour celui-ci, on part directement de l'ébauche. Celle-ci doit obligatoirement avoir été définie pour que l'algorithme puisse fonctionner.

Si on part des structures de données de l'algorithme standard, on remarque que, tout d'abord, nous avons besoin d'une structure supplémentaire contenant les erreurs du modèle, Cette structure étant définie sur chaque point de grille, et pour chaque pas de temps de la trajectoire du module. La question suivante a été de savoir si chaque module devait contenir les erreurs. En effet dans l'algorithme standard, les modules coût vont contenir les gradients.

On peut s'apercevoir ici que les actions B et E sont liables. En effet, on se souvient que η est utilisé seulement dans la boucle externe, mais pas dans la boucle interne.

Au niveau de C, on peut donc remplacer directement les valeurs de η par $p(t_i)$ dans l'adjoint. Le résultat de ces calculs sera stockés dans les gradients. Ensuite, une

fonction copiera les gradients dans les tableaux correspondant d'erreurs modèle, afin qu'ils soient cumulés par le tangent linéaire.

En effet, comme nous l'avons montré précédemment la routine D nécessite de cumuler les $p(t_i)$. Or il s'avère que selon la formule IV.36, nous avons seulement à multiplier $p(t_i)$ par Q_i pour obtenir l'erreur modèle. Nous pouvons donc garder $p(t_i)$ dans le tableau *ModelError* des modules.

Contrairement à l'algorithme standard, nous avons besoin ici de passer au minimiseur une structure correspondant à l'espace des observations, au lieu de l'espace des états, car nous travaillons sur le problème dual. Ceci signifie que la structure que nous devrons passer ne peut pas être de taille fixe. En effet, le nombre d'observations est fixé dynamiquement au niveau du fichier interprété. On pourrait éventuellement préparer un tableau contenant le nombre maximal d'observations, mais ceci occuperait en permanence de la mémoire, et le nombre total d'observations d'un problème peut être variable.

IV.2.10 Modifications du générateur

L'implémentation de cette fonctionnalité n'a pas nécessité de modification lourde du comportement du générateur. Il m'a toutefois fallu effectuer les modifications suivantes :

- modification de la grammaire ANTLR : ajout de l'option *O_DUAL*. Si celle-ci est activée, alors l'ensemble des structures et routines spécifiques au dual est alors généré,
- modification de la fonction *feedward* (au niveau de la génération du code des trajectoires), pour ajouter les méthodes *forward_dual* et *tanlin_dual* (points A et B).
- modification du template *Y$Project.cpp* : ajout des fonctionnalités nécessaires dans l'interpréteur (lancement d'une assimilation duale, fonction d'entrée/sortie sur les erreurs modèle),
- modification du fichier généré *Y1$Project.cpp* : ajout des erreurs modèles à chaque module concerné,
- ajout d'une classe *RunModelDual* : celle-ci contient les différentes boucles, routines génériques, et appels des itérateurs de trajectoires dans l'ordre indiqué par l'algorithme, ajout de la structure dynamique à passer au minimiseur,

- modification de la structure de *Dyniob.h* : l'arborescence des observations doit pouvoir contenir les variables m, l'innovation $d' = d - H K_1 \eta$, et la valeur courante pour chaque pas de temps supérieur à 1 de $\boldsymbol{H K_0} \delta x_0 + \boldsymbol{H \delta x}$.
- ajout de la classe *ObservationsComputing* : celle-ci permet de stocker les observations et d'effectuer facilement des calculs dans leur espace. Cette classe est destinée à remplacer le système précédent s'appuyant sur des listes chaînées.
- implémentation d'une routine de calcul de la fonction coût dans la nouvelle structure.

La première implémentation de la nouvelle structure nécessitait de s'appuyer sur les structures de modules. J'en ai donc profité pour ajouter les classes *Model* et *Space* au niveau des programmes générés.

IV.2.11 Algorithme dual d'un projet généré

On peut découper l'algorithme en deux parties :

- la boucle externe, implémentée au niveau de la fonction *iterateLoop*
- la boucle interne, implémentée au niveau de la fonction *basicIteration*

La boucle externe est décrite dans la figure IV.2.

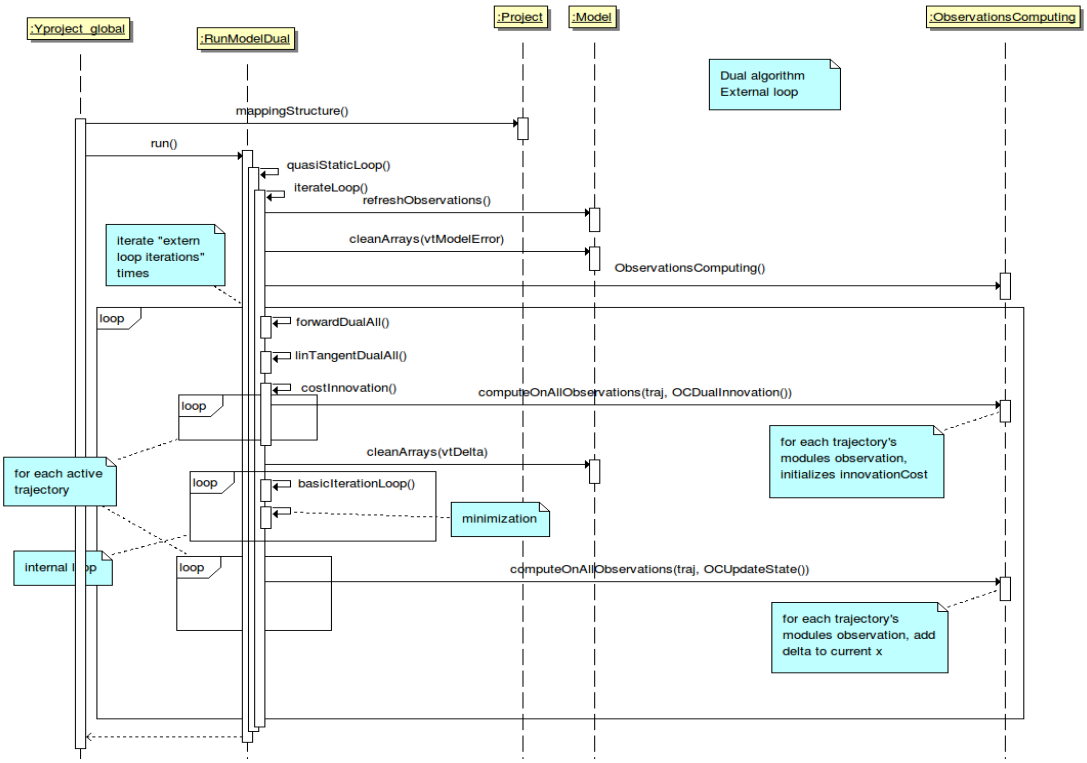

Figure IV.2: Diagramme de séquence de la boucle externe de l'algorithme dual

On peut retrouver dans cette figure les méthodes suivantes :

– *quasiStaticLoop* est l'implémentation d'une itération d'algorithme quasi-statique (présenté dans la partie IV.1)

– *iterateLoop* contient l'ensemble de l'itération externe,

– *refreshObservations* prépare la structure interne de gestion des observations,

– *cleanArrays* est une méthode générique permettant de remettre à zéro les tableaux passés en paramètre,

– une instance de la classe *ObservationsComputing* est crée.

– *forwardDualAll* est la routine interne permettant d'effectuer une passe avant cumulant les erreurs modèles.

– *linTangentDualAll* est la routine interne permettant d'effectuer un tangent linéaire cumulant les erreurs modèles.

– *costInnov ation* lance le calcul de d' sur l'ensemble des trajectoires, ce en utilisant les valeurs préalablement calculées. Ces calculs s'effectuent dans l'espace des observations.

– *computeOnAllObservations*(*traj, OCDualInnovation()*) effectue des calculs sur l'ensemble d'une trajectoire. Une instance de classe est passée en paramètre pour indiquer le calcul à effectuer (ici *OCDualInnovation*). Il s'agit ici des cumuls nécessaires au calcul de d' sur une trajectoire donnée.

– *basicIterationLoop* contient la boucle interne. Cette méthode est lancée à chaque itération de minimisation, quelque soit la méthode utilisée (m1qn3 ou algorithme standard).

– *computeOnAllObservations*(*traj, OCUpdateState()*) effectue les cumuls de fin d'itération de boucle externe nécessaires pour obtenir les nouvelles valeurs de x_0^t.

La boucle interne est décrite dans la figure IV.3. Elle contient les appels aux méthodes suivantes :

– *cleanArrays*(*vtGradient*) : nettoyage préalable du tableau des gradients,

– *backwardDualAll* : adjoint rétropropageant les variables m, pour le calcul de $p(t_i)$,

– *cleanArrays*(*vtModelError*) : nettoyage des erreurs modèle,

– *coefCovarianceMatrix* : multiplication par B et par Q des $p(t_i)$,

– *copyGradientToModelError* : copie des variables $p(t_i)$ présentes dans le tableau des gradients vers le tableau des erreurs modèle (pour réutilisation au niveau de la boucle externe),

– *partiallyCleanGradients* : nettoyage du tableau des gradients, en gardant δx_0,

– *linTangentAll* : tangent linéaire partant de δx_0 dans le but de calculer δx,

– *costLinearTangent* : cumul de la fonction coût avec résultats préalablement calculés,

– *copyGradientToDeltaX* : récupération des gradients dans le tableau des delta X,

– *cleanArrays*(*vtGradient*) : nettoyage du tableau des gradients,

– *linTangentDualAll*() : tangent linéaire cumulatif partant de δx_0 dans le but de calculer $\boldsymbol{H}\boldsymbol{K}_0\delta x_0$,

– *costSum*() : sommage des différents éléments pour le calcul final des deux élements de la fonction coût.

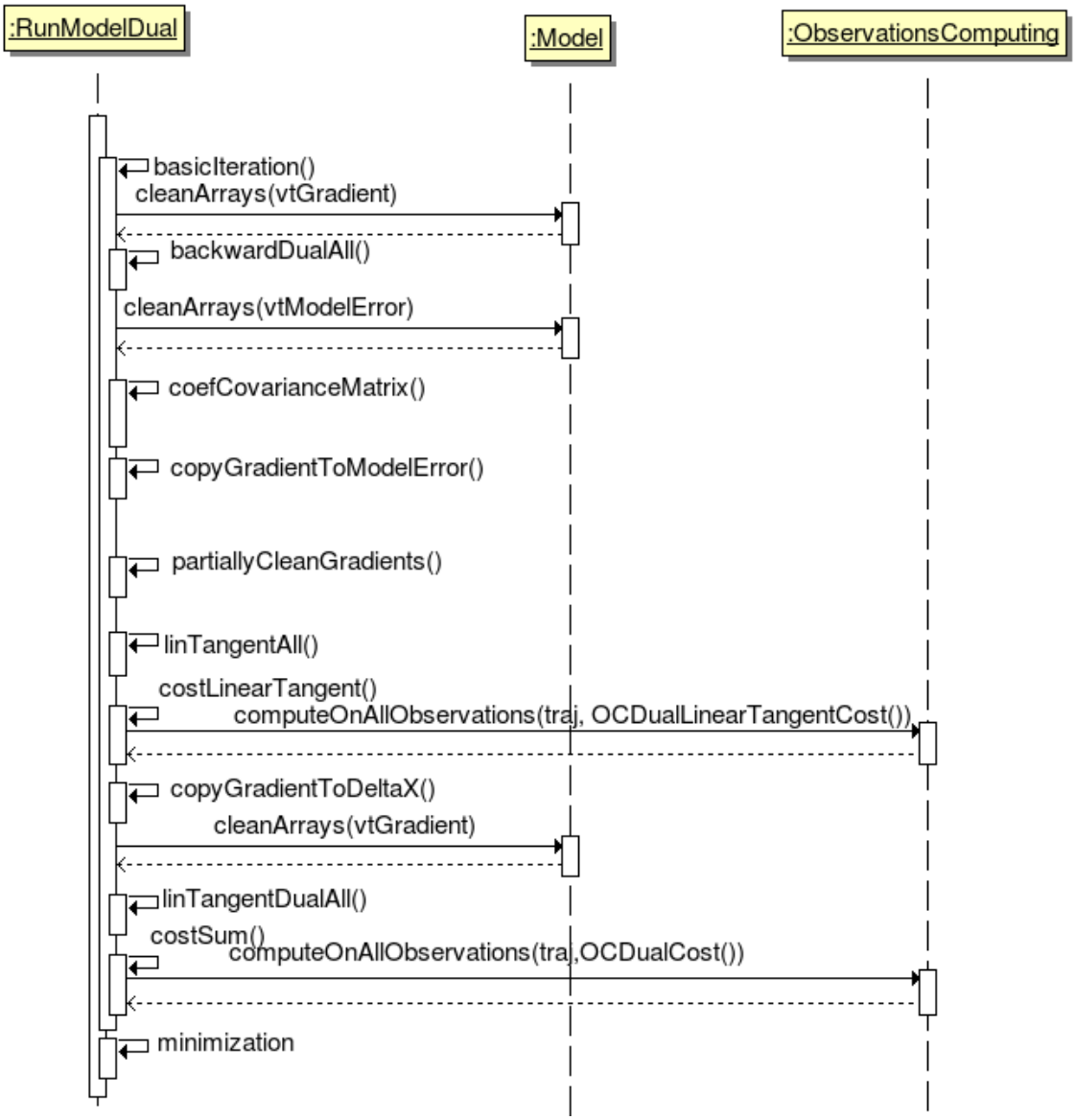

Figure IV.3: Diagramme de séquence de la boucle interne de l'algorithme dual

On retrouve bien dans le code implémenté l'ensemble des routines nécessaires à l'algorithme de l'assimilation.

IV.2.12 Tests

Les tests ont été effectués sur le modèle du Shallow Water. Le but est ici de vérifier que l'implémentation de l'algorithme a été bien réalisée, et que ce dernier permet une

amélioration par rapport à l'utilisation de l'algorithme normal d'assimilation variationnelle, ce pour des cas simples.

IV.2.12.1 Shallow Water

Le modèle du Shallow-Water décrit dans les parties précédentes est suffisamment simple pour que nous puissions comprendre la teneur des paramètres, et les faire évoluer. De plus l'ensemble des modules est déjà prêt. Il s'agit ici d'un modèle non linéaire, nous permettant de vérifier le bon fonctionnement de l'algorithme dans ce type de cas. L'espace est ici en 2 dimensions, de taille 50×50. La taille temporelle du problème est de de 51 pas de temps.

Pour rappel, le modèle du Shallow Water décrit la mécanique d'écoulement d'une étendue d'eau. Les valeurs évoluant dans le modèle sont la hauteur de l'eau, ainsi que les vitesses horizontales et verticales. Les paramètres physiques sont en particulier :

- la gravité réduite
- le facteur de dissipation
- la hauteur moyenne d'eau

Nous avons repris l'exemple de départ (tel qu'il est décrit dans la partie II.3.1), où nous devions retrouver une colonne d'eau de 15 mètres à partir des observations de la hauteur sur le pas de temps final, ainsi que d'une ébauche/first guess de 10 mètres. La distance quadratique entre les deux ensembles est de $0{,}1963$. Les algorithmes devront donc forcément améliorer ce rapport pour pouvoir être intéressants.

Le premier test a consisté à rajouter un bruit sur l'ensemble des pas de temps, généré à l'aide d'une loi normale d'écart type 0,31. Pour simplifier la mise en place, le même bruit est utilisé pour chaque pas de temps. Ceci peut poser des problèmes, cependant cela nous permet également de vérifier la robustesse de l'algorithme.

Dans la figure IV.4, il est déjà possible de remarquer « à l'œil nu » que les résultats fournis par l'assimilation variationnelle à contrainte faible sont meilleurs (les vaguelettes sont plus présentes dans ce dernier).

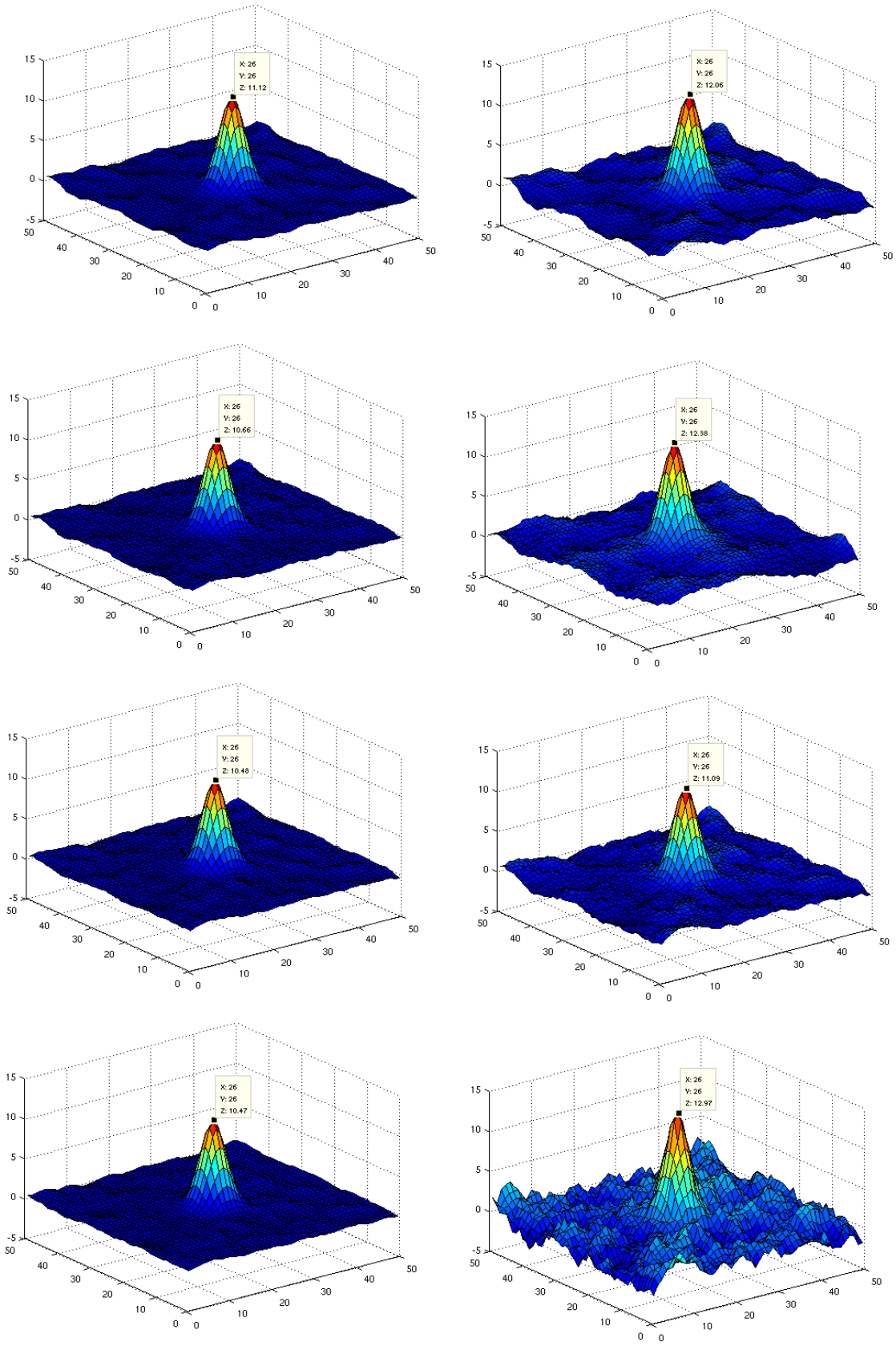

Figure IV.4: Courbes représentants les hauteurs obtenues en fonction des coordonnées dans l'espace pour les résultats de l'algorithme standard et de l'algorithme dual avec une erreur modèle couvrant l'ensemble de l'espace.
De haut en bas, de 4 à 1 pas de temps d'observation. La fenêtre blanche représente les coordonnées et la hauteur du point central de l'espace.

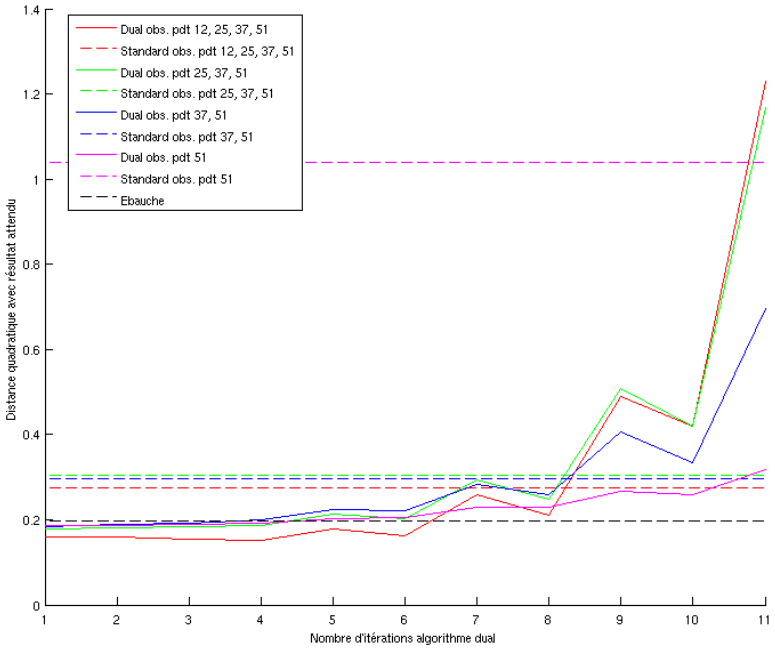

Figure IV.5: Résultats algorithme dual en fonction du nombre d'observation pour une erreur modèle couvrant l'espace complet.
Les lignes en pointillés représentent les résultats de l'algorithme à contrainte forte, les lignes pleines ceux de l'algorithme dual. Des essais ont été réalisés avec plusieurs nombres d'observations, chaque couleur représentant un nombre différent. La distance quadratique entre l'ébauche et le résultat est représenté par la droite noire en pointillés. Les résultats de l'algorithme standard sont représentés en traits pleins.

Les résultats obtenus par l'algorithme dual diffèrent selon le nombre d'itérations externes. Dans la figure IV.5, on peut remarquer qu'en général c'est à l'itération 3 qu'on obtient la meilleure erreur. Les erreurs baissent jusqu'à l'itération 3 ou 4, puis ensuite remontent en oscillant de manière exponentielle. Il est à noter toutefois que les résultats sont meilleurs que l'algorithme standard, ce pour un faible nombre d'itérations.

Le deuxième test a consisté à centrer l'erreur modèle au milieu de l'espace. En effet, au vu de la configuration de la gaussienne, les hauteurs sont égales à zéro près du bord de l'espace, et peuvent être jusqu'à 15 mètres au centre de l'espace. L'effet des erreurs modèle sur les résultats est donc bien plus grand sur les bords que sur le centre. Le principe que j'ai pris a été de prendre une erreur modèle nulle partout sauf sur un carré de 10 par 10 au centre de l'espace. L'écart type utilisé reste le même que pour la première expérience.

On peut retrouver dans la figure IV.6 les résultats obtenus. La distance quadratique entre les résultats obtenus et la hauteur cherchée est bien plus basse dans ce cas. Ceci est logique, vu qu'il y a beaucoup moins d'erreur de modèle. Les performances de l'algorithme dual restent supérieures à celles de l'algorithme standard.

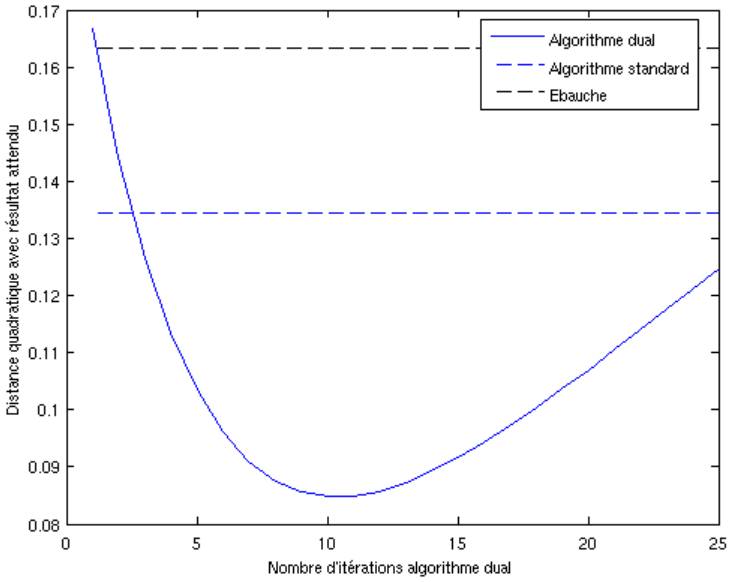

Figure IV.6: Performance de l'algorithme dual avec erreur modèle centrée sur la gaussienne.
La courbe noire en pointillés représente la distance quadratique de l'ébauche avec les paramètres à retrouver. La droite bleue en pointillés représente la distance quadratique de la valeur obtenue par l'algorithme standard avec les paramètres à retrouver. La courbe bleue en traits pleins représente la distance quadratique de l'algorithme dual avec les paramètres à retrouver, en fonction du nombre d'itérations externe.

Pour le troisième test, nous avons décidé de ne plus ajouter d'erreur modèle au modèle initial, mais plutôt de modifier un des paramètres physiques de celui-ci, afin d'avoir un cas plus proche de la réalité. Le principe était ici de réaliser une passe avant du modèle avec toujours pour initialisation une gaussienne de 15 mètres avec une gravité réduite de 0,01, comme pour les tests précédents. Cependant, pour l'assimilation, on utilise une gravité modifiée.

Pour ce cas, il m'a fallu calculer la distance entre les sorties du modèle avec les deux gravités. Celle-ci m'a donné une estimation du coefficient à appliquer à la matrice Q.

Après avoir effectué plusieurs expérimentations avec différentes valeurs de gravité réduite, les résultats obtenus sont décrits dans le tableau IV.2.

Gravité réduite de l'expérience	Distance globale avec modèle parfait	Distance observations avec modèle parfait	Erreur algorithme standard	Erreur algorithme dual
0,005	7,50E-004	4,40E-003	3,00E-002	3,60E-003
0,02	1,70E-003	8,60E-003	3,38E-002	2,72E-002
0,03	4,20E-003	2,19E-002	6,64E-002	6,22E-002
0,04	6,00E-003	3,20E-002	9,46E-002	8,75E-002

Tableau IV.2: Distance quadratique entre les résultats obtenus par les différents algorithmes et la réalité pour plusieurs valeurs de gravité réduite.

La figure IV.7 montre les résultats obtenus selon le nombre d'itérations. Les résultats semblent moins bons, ce qui peut éventuellement s'expliquer par la différence entre la nature des erreurs. En effet, pour les deux premières expérimentations, l'erreur modèle était générée à partir d'une loi normale, ce qui fait partie des postulats de base concernant l'erreur modèle. Avec une modification de la gravité réduite, nous avons ici une erreur dont la moyenne n'est pas obligatoirement nulle, et dont la distribution ne respecte pas obligatoirement une loi normale. Ceci peut expliquer la raison pour laquelle les résultats sont moins bons que pour la deuxième expérience. Notons toutefois que nous obtenons toujours de meilleurs résultats qu'avec l'algorithme standard, même si ceux-ci sont extrêmement proches.

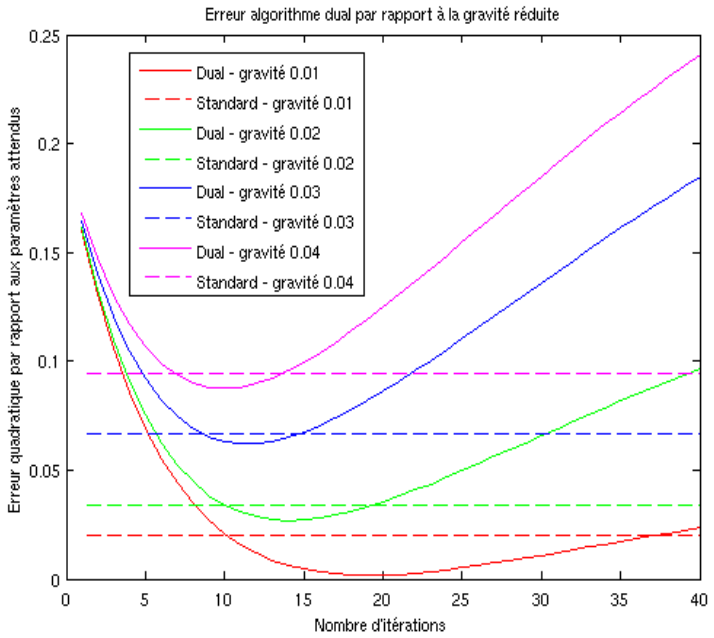

Figure IV.7: Performance de l'algorithme dual avec gravité réduite modifiée
Les traits pleins représentent ici les performances de l'algorithme standard. Les courbes en pointillés représentent les performances de l'algorithme dual en fonction du nombre d'itérations externes.

IV.2.13 Evolutions restant à réaliser

La gestion des opérateurs de variance/covariance, de l'erreur modèle semble être très importante. Elle devrait en effet permettre d'approcher d'une façon bien plus précise les paramètres attendus. En effet, l'erreur modèle ne se comporte pas de la même manière en chaque point de grille, et selon chaque pas de temps. Or, actuellement pour l'algorithme dual, il est possible seulement d'utiliser des coefficients pour ces matrices. Celles-ci ne peuvent être alors que de la forme λI où λ est le coefficient, et I la matrice identité. Ceci provoque le fait que les coefficients sont identiques en chaque point, ce qui est loin de la réalité des modèles (comme nous l'avons vu avec le Shallow Water).

Une explication de la dégradation des résultats après un extremum peut être du à la nature du fonctionnement de l'algorithme dual. En effet, la minimisation de la version duale d'un problème doit se faire à l'optimum. Les résultats obtenus avant celui-ci peuvent ne pas être

Il reste également à effectuer des analyses plus poussées sur les performances de l'algorithme. En effet, les tests effectués ici ont été en général réalisés avec des valeurs relativement arbitraires pour ces différentes matrices. Une vérification plus approfondie du générateur est donc à effectuer.

V Optimisation des programmes générés

Historiquement, le calcul numérique a toujours été un grand consommateur de performances. La montée en charge des processeurs a permis de modéliser des problèmes de plus en plus complexes, mais leur taille a également augmenté en parallèle. Or, dernièrement, comme le signalent Chellapa et al. [29], les fabricants ont atteint une limite technique au niveau de la gravure. Ceci s'est manifesté par une stagnation des fréquences des différents processeurs construits durant les 5 dernières années. La loi de Moore reste cependant respectée grâce à l'augmentation du nombre de coeurs dans les différents processeurs, ainsi qu'à l'ajout de fonctionnalités supplémentaires comme par exemple la vectorisation. Toutefois, pour tirer partie de ces nouvelles instructions -qui n'existaient pas auparavant- il est souvent nécessaire de faire évoluer les programmes.

Yao, se proposant de fournir une aide à la mise au point de modèles numériques, il doit donc fournir des performances permettant son utilisation. En l'occurrence, deux

problèmes ont été remontés par les utilisateurs à propos des programmes générés par Yao. Ils sont liés en particulier à la consommation mémoire d'un projet, et à sa rapidité d'éxecution. Un projet de parallélisation du code de Yao est en cours d'implémentation pour pallier à ce problème. Cependant les performances du code séquentiel ne sont optimales, et c'est certains de ces points que nous aborderons dans cette partie.

V.1 Pistes d'optimisation

Malgré les fonctions d'optimisation de plus en plus poussées des compilateurs, les différents algorithmes se voulant « optimisés » se doivent de tenir compte des particularités d'une plateforme. En effet, les compilateurs ne sont pas encore capables d'optimiser totalement un code source pour une architecture donnée. Dans le cas de programmes non optimisés, on peut parfois observer un écart substantiel vis à vis de la vitesse qu'on pourrait en attendre.

En général, les débits de lecture/écriture d'un disque dur à plateaux, de la mémoire vive et d'un processeur sont très éloignées (un facteur 100 environ entre chacune). Pour remédier à ce problème, il existe plusieurs niveaux de mémoire dite « mémoire cache » à l'intérieur d'un processeur. Ce type de mémoire, bien plus onéreux que la mémoire conventionnelle est présente en quantité bien moindre par rapport à celle-ci (vis à vis de son prix), mais est également beaucoup plus rapide. Il peut y avoir de 1 à 3 niveaux de mémoire cache selon les architectures. Les écarts de performance sont en général d'à peu près d'un facteur 10 entre chaque niveau, ainsi qu'entre le processeur et le premier niveau. Un programme bien optimisé utilisera le moins possible dans ses boucles de traitements des portions de code utilisant la mémoire centrale. Toute opération se répétant sur des cases mémoire déjà chargées en cache doit au maximum s'appuyer sur ce dernier. Une partie de ces actions est gérée automatiquement par le processeur.

Le principe général est que lorsque le processeur a besoin d'accéder à une case mémoire, si la donnée en question n'existe pas en mémoire cache, celle-ci est alors chargée. Si elle se trouve déjà en mémoire cache, elle est directement utilisée par le processeur, ce à une vitesse bien supérieure à celle de la mémoire centrale. D'une manière similaire, les écritures se font d'abord en mémoire cache, et sont ensuite réécrites en mémoire centrale.

La mémoire cache est basée sur le principe de localité. Selon Wikipedia [30], il existe deux principes de base :

- la localité temporelle : une donnée accédée à un instant t a des chances d'être accédée à nouveau à un temps ultérieur ;
- la localité spatiale : il y a souvent une forte probabilité qu'un programme ayant besoin d'une donnée à un emplacement mémoire donné ait besoin d'une autre donnée à un emplacement proche.

A partir de ces deux principes, on peut déduire deux autres localités : la localité équidistante résultant de l'utilisation de tableaux et de boucles (les indexes des tableaux étant prévisibles linéairement), et la localité de branche (le résultat des sauts conditionnels est souvent répétable, et peut être ainsi optimisé).

Dans le cas où un chargement depuis la mémoire vive jusqu'à la mémoire cache a lieu, on a affaire à un défaut de cache. Ce type de chargement est évidemment très lent (il se fait à la vitesse de la mémoire vive), et doit survenir le moins souvent possible dans un programme. Il en existe quatre types :

- les défauts de cache obligatoires, dans le cas du premier accès à une donnée,
- les défauts de cache capacitifs, dans le cas où la taille du cache est dépassée par le,
- les défauts de caches conflictuels, dans le cas où deux adresses du cache correspondent au même emplacement mémoire,
- les défauts de cohérence, dans les cas multi processeur.

Il peut être intéressant d'effectuer le profilage d'un programme pour vérifier sa quantité de défauts de cache, ventilée par fonction. C'est ce que nous étudierons dans la partie suivante, après une estimation de la durée d'exécution du programme.

V.2 Analyse du code de YAO

Les tests ont été réalisés sur l'implémentation du Shallow Water telle que décrite dans la partie II.3.1. Il s'agit ici d'une simple expérience jumelle telle que décrite dans les parties précédentes. J'ai repris ici le source généré par Yao pour ce projet, et j'y ai apporté quelques modifications. Les dimensions de l'espace ont été rendues paramétrables dans ce projet via l'ajout de directive de précompilation « define ». Celles ci sont modifiables via le makefile créé pour le projet de test. Trois configurations ont été préparées : des espaces

correspondant à des grilles de 50×50 , de 100×100 et de 200×200 . Le nombre de pas de temps de la trajectoire est de 100. Le nombre d'itérations maximal de la descente du gradient est fixée à 50 pour ces configurations. Les opérations indiquées dans le fichier d'interprétation, et donc exécutées par le programme, sont dans l'ordre suivant :

- l'initialisation des valeurs
- le lancement d'une directive *forward* (passe avant du modèle), dans le but d'obtenir les valeurs à comparer
- écriture des résultats dans un fichier et récupération des valeurs résultantes comme ébauche
- légère perturbation des paramètres en entrée
- phase d'assimilation complète à l'aide du minimiseur m1qn3
- écriture des résultats dans un autre fichier

Il est faut noter ici que l'opération d'optimisation a été réalisée avant la modification de structure décrite dans la partie III.3.

V.2.1 Présentation des outils

De nombreux outils de profilage existent pour les plateformes Unix/Linux. Le principe de ces outils peut être d'ajouter au programme des fonction de simulation, ou de simuler le comportement d'un processeur. J'ai décidé d'utiliser les outils *gprof* et *valgrind* pour évaluer les performances et la quantité de défauts de cache.

V.2.1.1 Gprof

L'utilitaire *gprof* [31] est inclus en standard dans la plupart des distributions Unix/Linux. Il est couplé à la directive « -pg » de gcc qui ajoute des informations dans l'exécutable [32] permettant l'analyse précise des appels de fonctions lors d'une exécution.

V.2.1.2 Valgrind

Gprof est un outil très utile, cependant il reste très basique et ne donne pas d'informations concernant l'utilisation de la mémoire cache. Après une courte recherche sur les différents outils utilisables, mon choix s'est porté sur l'outil *Valgrind*.

Valgrind est un outil permettant l'analyse de l'utilisation de la mémoire cache, la détection de fuites de mémoire, et le profilage de code. Selon son manuel, il est constitué des modules suivants, en particulier :

- *MemCheck*, un détecteur d'erreurs d'accès à la mémoire. Il permet de déterminer les fuites de mémoire, les violation d'accès, les utilisations de valeurs non initialisées, etc.
- *Cachegrind* est un outil simulant le couple processeur/mémoire afin de déterminer les défauts de cache de facon relativement fine en émulant le fonctionnement d'un processeur.
- *Callgrind* reprend les fonctionnalités de *Cachegrind* en y rajoutant une vision dynamique des appels des fonctions, ceci permettant d'observer de manière arborescente les statistiques.
- *Helgrind* et *DRD* permettent de détecter les erreurs survenant sur les threads, comme par exemple les situations de compétition (data race), les interblocages, etc.
- *Massif*, *DHAT*, et *PtrCheck* sont des outils d'analyse de la consommation mémoire.

Comme l'indique Nethercote [32], *Cachegrind* est un bon outil pour évaluer la mémoire cache. Son principe est d'émuler le fonctionnement d'un processeur avec plusieurs niveaux de cache mémoire. Le but n'est pas de l'imiter à la perfection, mais d'en reproduire suffisamment le fonctionnement pour détecter les parties à optimiser d'un programme. J'ai également utilisé le programme *Kcachegrind* [33] afin d'avoir une visualisation simple des performances obtenues. Ce dernier est une interface graphique permettant d'observer visuellement les analyses effectuées par les modules *Cachegrind* ou *Callgrind*. Il est possible grâce à cet outil d'observer ligne par ligne les statistiques, tout en ayant des rendus visuels plus intuitifs que ceux fournis en modes texte.

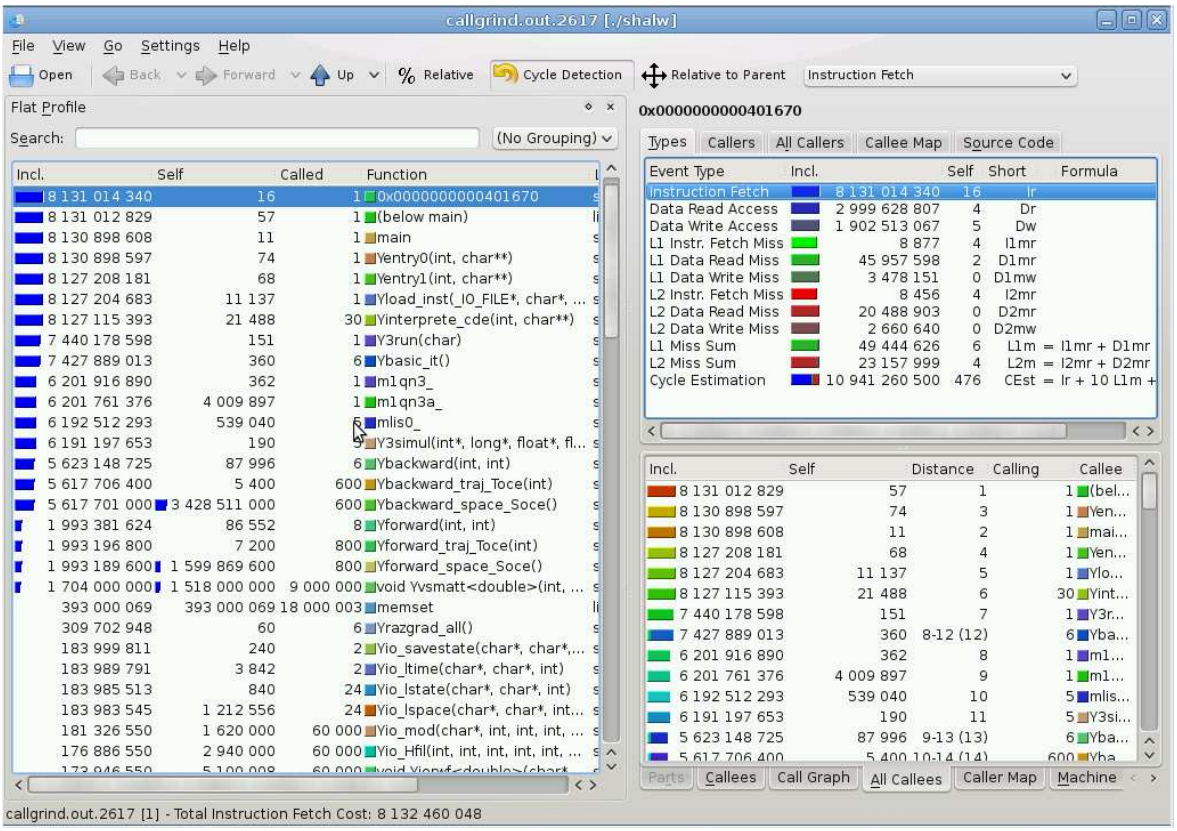

Figure V.1: Capture d'écran de *Kcachegrind*.

On peut retrouver dans la figure V.1 un exemple d'écran de *Kcachegrind*. Les mesures proposées à l'analyse sont celles de *Callgrind*, en l'occurence :

– le nombre d'instructions,

– le nombre d'accès mémoire en lecture, et en écriture,

– le nombre de défauts de cache sur les instructions, en lecture, et en écriture, pour chaque niveau de cache,

– une estimation du nombre de cycles processeur calculé à partir des statistiques précédentes.

L'ensemble de ces mesures m'a permis de mesurer l'impact des modifications réalisées au niveau de la mémoire cache.

V.2.2 Résultats du profilage

V.2.2.1 Gprof

La figure V.2 est un extrait de profil plat obtenu pour un espace de taille 100×100. Un profil plat montre la durée totale de temps employée par chaque fonction, son nombre d'appels, ainsi que le temps par appel.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 41.25     44.76    44.76     5200     0.01     0.01  Ybackward_space_Soce()
 22.42     69.09    24.33 312000000     0.00     0.00  void Yvsmatt<double>(int, int, int, double*, double*, double*)
 21.06     91.94    22.85     5400     0.00     0.01  Yforward_space_Soce()
  1.69     93.77     1.83 52000000     0.00     0.00  Hphy::backward(double, double, double, double, double)
  1.68     95.59     1.82 54000000     0.00     0.00  Hphy::forward(double, double, double, double, double)
  1.28     96.98     1.39 54000000     0.00     0.00  Uphy::forward(double, double, double, double, double, double,
double)
  1.25     98.34     1.36 52000000     0.00     0.00  Uphy::backward(double, double, double, double, double, double,
double)
  1.19     99.63     1.29 54000000     0.00     0.00  Vphy::forward(double, double, double, double, double, double,
double)
  1.11    100.83     1.21 52000000     0.00     0.00  Vphy::backward(double, double, double, double, double, double,
double)
  1.09    102.01     1.18       52     0.02     0.02  Yrazgrad_Hfil()
  1.03    103.13     1.12       52     0.02     0.02  Yrazgrad_Hphy()
  0.88    104.09     0.96       52     0.02     0.02  Yrazgrad_Uphy()
  0.62    104.77     0.68 54000000     0.00     0.00  Vfil::forward(double, double, double)
  0.62    105.44     0.67 54000000     0.00     0.00  Hfil::forward(double, double, double)
  0.53    106.01     0.57 54000000     0.00     0.00  Ufil::forward(double, double, double)
  0.41    106.46     0.45       52     0.01     0.01  Yrazgrad_Vfil()
  0.41    106.90     0.44       52     0.01     0.01  Yrazgrad_Vphy()
  0.35    107.28     0.38       52     0.01     0.01  Yrazgrad_Ufil()
  0.26    107.56     0.28 52000000     0.00     0.00  Hfil::backward(double, double, double)
  0.23    107.81     0.25 52000000     0.00     0.00  Vfil::backward(double, double, double)
  0.22    108.04     0.24 52000000     0.00     0.00  Ufil::backward(double, double, double)
  0.10    108.15     0.11                             sdot_
  0.09    108.25     0.10    10000     0.00     0.00  Vphy::Vphy()
  0.05    108.30     0.05   520000     0.00     0.00  Ywishdiff_all(char*, int, int, int, int, int, double)
  0.05    108.35     0.05   520000     0.00     0.00  Ycostwishdiff_all(char*, int, int, int, int, int)
  0.04    108.39     0.04                             m1qn3a_
  0.03    108.42     0.03       51     0.00     0.00  Y3getstate_Hfil(float*)
  0.02    108.44     0.02                             ctlcan_
  0.02    108.46     0.02                             dd_
  0.01    108.47     0.01    60000     0.00     0.00  void Yiorwf<double>(char*, int, int, int, int, int, double*,
double)
  0.01    108.48     0.01     5252     0.00     0.00  Ybackward_elect()
  0.01    108.49     0.01       54     0.00     0.54  Yforward(int, int)
  0.01    108.50     0.01       52     0.00     0.00  Y3valgrad_Hfil(float*)
  0.01    108.51     0.01        1     0.01     0.01  xvitgeo()
  0.01    108.52     0.01                             Yio_Ufil(int, int, int, int, int, double)
```

Figure V.2: Résultats du profilage effectué par gprof pour une grille de 100x100.

Les colonnes présentées sont de gauche à droite le pourcentage de temps de chaque procédure, le temps cumulé, la durée, le nombre d'appels, la durée par appel, et le nom de la fonction.

Il faut remarquer que les opérations des modules (*module* ::forward et *module* ::backward) ne sont pas comptabilisées dans les méthodes de passe avant (*Yforward_space_soce*) et de rétropropagation (*Ybackward_space_soce*). Au vu de l'architecture du code source de ces deux procédures, on peut en conclure que le temps est surtout pris par des accès de données. En effet, on retrouve surtout des affectations ou des appels de fonctions inclus dans les boucles imbriquées.

On remarque ici que les passes avant sont appelées plus de fois que les rétropropagations. Ceci est du à la première passe avant permettant de récupérer les valeurs à comparer pour l'apprentissage. On considérera ici qu'entre les différentes configurations, le nombre d'appels des procédures *Yforward_space_soce* et *Ybackward_space_soce* est

similaire et que, malgré les légères différences, il est possible de les comparer. En effet, nous souhaitons ici surtout obtenir des indications d'ordre général sur les points à optimiser du programme.

A partir du profil plat on peut déjà observer plusieurs choses :

– la procédure de rétropropagation (*Ybackward_space_Soce*) prend 40 à 45% du temps

– la multiplication vecteur-matrice transposée (*Yvsmatt*) prend plus de 20 à 25% du temps. Cette fonction est utilisée par la procédure *Ybackward_space_Soce*, celle ci y faisant appel 240.000 fois par lancement.

– la procédure de passe avant (*Yforward_space_Soce*) prend 20% du temps.

– les méthodes (*module* ::forward) et (*module* ::backward) correspondent au calcul de la passe avant et au calcul du jacobien du module. Leur total avoisine les 10 % du temps de traitement. Ces méthodes sont appelées respectivement par les méthodes *Yforward_space_Soce* et *Ybackward_space_Soce*, mais ne sont pas comptabilisées dans leur temps.

– les méthodes *Yrazgrad_module* sont appelées par *Ybackward_space_Soce*. Elles prennent moins de 5%.

– le reste des fonctions (toutes n'apparaissent pas sur les profils) représentent une durée mineure.

Le profil plat de l'application Shallow Water sur une grille de 200×200 donne des résultats sensiblement identiques à ceux de la grille de 100×100 au niveau des proportions.

On voit ici que les actions à réaliser sont en particulier sur les fonctions *Ybackward_space_Soce*, *Yforward_space_Soce*, et *Yvsmatt* (multiplication vecteur-matrice transposée), ces trois fonctions totalisant à elles seules plus de 80% du temps de traitement du programme.

V.2.2.2 Valgrind

J'ai commencé par effectuer un test à l'aide *Memcheck*. L'analyse à l'aide de cet outil montre peu de points à améliorer. Aucune fuite mémoire n'est présente dans le programme.

L'analyse de *Cachegrind* s'avère bien plus intéressante. En effet les résultats obtenus sont décrits dans le tableau V.1.

Procédure	Indicateur « cycle estimation »	Lectures en mémoire	Ecritures en mémoire	Instructions appelées	Défauts de cache de niveau 1	Défauts de cache de niveau 2
Ybackward_space_Soce (pourcentage)	45,1	40,44	43,95	42,16	51,87	53,99
Yforward_space_Soce (pourcentage)	21,05	23,8	22,07	19,67	31,59	23,62
Yvsmatt (pourcentage)	16,5	20,35	12,3	18,67	9,1	10,46
Memset (pourcentage)	3,59	1,2	5,52	4,83	0[3]	0
Total forward et backward de chaque module (pourcentage)	7,63	9,28	10,17	8,25	4,42[4]	6,13
Total programme (valeur)	*10943617241*	*2999982821*	*1902630491*	*8132460051*	*49456299*	*23165942*

Tableau V.1: Résultats de l'analyse de Callgrind sur le projet Shallow Water (taille 100×100).

Quelques points sont à préciser concernant le tableau V.1 :

- Les défauts de cache présentés dans le tableau représentent la somme des défauts concernant les appels d'instructions, les lectures et les écritures, ce pour chaque niveau.
- L'indicateur « *Cycle estimation* » donne une estimation empirique du coût total des défauts de cache. Il représente la somme du nombre d'instructions, le total de défauts de cache de niveau 1 multiplié par 10 et le total de défauts de cache de niveau 2 multiplié par 100. En effet, les performances de la mémoire cache de niveau 1 sont en général 10 fois supérieures à celles de la mémoire cache de niveau 2.

En observant tout d'abord les valeurs de l'indicateur « *cycle estimation* », j'ai tout d'abord remarqué que les procédures de passe avant, passe arrière et de multiplication de matrices sont celles qui ont les pourcentages les plus élevés dans les 3 cas, et ce toujours dans le même ordre, quelque soit la taille d'espace choisie. Ces 3 procédures sont également les plus longues à traiter comme nous l'avons vu dans la partie précédente avec *gprof*.

[3] La procédure memset ne faisait pas partie des statistiques pour les défauts de cache. Ceci est probablement dû au fait qu'elle fait partie d'une bibliothèque extérieure n'ayant pas été compilée avec les informations de débogage.

[4] Étrangement, seules les fonctions de passe avant ont été valorisées dans le compte rendu de Callgrind, les fonctions de calcul de jacobien n'ont pas été comptées.

Une analyse des rapports précis de Callgrind montre qu'au niveau du code source et des procédures *Ybackward_space_Soce* et *Yforward_space_Soce*, les défauts de cache surviennent dans les boucle de traitement de l'espace, au niveau de la lecture des valeurs de celui-ci. Il y a donc une action à réaliser à ce niveau.

V.3 Actions réalisées et performances

V.3.1 Optimisation de l'utilisation de la mémoire cache

Pour rappel, la structure d'origine des données des modules d'un programme généré est sous la forme décrite par la figure V.3 :

```
Hfil                *YHfil[YA1_Soce][YA2_Soce] ;
class          YaoHfil
{
          public:
               double       Ystate [YNBALLTIME_Toce] [YNBS_Hfil];
               double       Ygrad[YNBALLTIME_Toce][YNBS_Hfil];
               double       Yepsi[YNBS_Hfil];
               double       Ywish[YNBS_Hfil];

YaoHfil(){}
~YaoHfil(){}

};
```

Figure V.3: Exemple de structure d'un module (version d'origine)

Le problème ici est que dans le source généré actuel, l'ordre des boucles ne correspond pas à l'ordre de la structure. En effet, les boucles extérieures sont au niveau des sorties et du temps, les boucles intérieures du programme servant au parcours de l'espace en chaque point. Or, comme nous l'avons vu dans les parties précédentes, Yao traite toujours un espace complet, ce pour chaque pas de temps. L'utilisation du cache donc n'est pas optimale car au lieu de traiter les éléments consécutifs en mémoire, on traite ici des éléments non contigus. Ceci est clairement visible dans la figure V.4.

```
YA1=50; YA2=50; YA3=0;
Yi=-1; Yj=-1; Yk=-1; /* init des indices de maille: maj par la
boucle si valide*/
for(Yi=0; Yi<YA1_Soce; ++Yi)
for(Yj=0; Yj<YA2_Soce; ++Yj)
{
           Yting[0]=YS1_Hfil( Yi, Yj, YTemps-1);
           if (1==0 || Yi-1<0)
           Yting[1]=0;
           else
           Yting[1]=YS1_Uphy( Yi-1, Yj, YTemps-1);
           Yting[2]=YS1_Uphy( Yi, Yj, YTemps-1);
           Yting[3]=YS1_Vphy( Yi, Yj, YTemps-1);
           if (1==0 || Yj+1>YA2_Soce-1)
           Yting[4]=0;
           else
           Yting[4]=YS1_Vphy( Yi, Yj+1, YTemps-1);

           YHphy->forward( Yting[0], Yting[1], Yting[2], Yting[3],
Yting[4] );
          // Autres modules ...
}
```

Figure V.4: Code source du traitement d'un module (module Hphy)

Pour rappel, les variables *YS1_* présentées dans la figure sont des macros pointant sur les données du module correspondant. En l'occurrence, *YS1_Hfil(a,b,c)* pointe sur *Yhfil[a,b]->Ystate[c, 0]*.

La modification que j'ai effectué a consisté à intervertir l'ordre de la structure afin qu'elle corresponde mieux à l'ordre des boucles du programme généré.

La structure après optimisation est décrite dans la figure V.5.

```
Hfil              *YHfil;
class          YaoHfil
{
          public:
                  YREAL     Ystate[YNBALLTIME_Toce][YA1_Soce][YA2_Soce][YNBS_Hfil];
                  YREAL     Ygrad[YNBALLTIME_Toce][YA1_Soce][YA2_Soce][YNBS_Hfil];
                  YREAL     Yepsi[YA1_Soce][YA2_Soce][YNBS_Hfil];
                  YREAL     Ywish[YA1_Soce][YA2_Soce][YNBS_Hfil];

//:=========> Constructor - Destructor ============
YaoHfil(){}
~YaoHfil(){}

};
```

Figure V.5: Structure après optimisation

Ici, j'ai repris le code source généré décrit dans la partie précédente, et je l'ai adapté afin d'étudier le nouveau comportement. Cette modification a été relativement simple au vu du fait que la plupart des accès aux éléments sont fait via des directives *#define*.

Il faut également noter qu'au lieu d'avoir un tableau d'instances de *Yhfil*, on a désormais une seule instance de *Yhfil*, contenant des tableaux dont les indexes sont dans l'ordre le temps, l'axe *X*, l'axe *Y*, puis la sortie du module. Concernant les derniers indexes (espace et sortie) la question de leur ordre s'est posée. Notre choix s'est porté sur la définition des sorties en dernier.

Après avoir effectué ces modifications dans le projet de test, dès le premier lancement, j'ai pu constater une amélioration sensible du temps de traitement. Les analyses par les outils *gprof* et *valgrind* sont détaillées dans les parties suivantes.

V.3.1.1 Analyse par Gprof

Les résultats donnés par une analyse *gprof* sur le projet de test avec une grille de 100x100 montrent ce gain de temps substantiel, comme le montre la figure V.6.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 36.99     21.55    21.55     5200     0.00     0.01  Ybackward_space_Soce()
 26.50     36.99    15.44 312000000     0.00     0.00  void Yvsmatt<double>(int, int, int, double*, double*, double*)
  6.95     41.04     4.05     5400     0.00     0.00  Yforward_space_Soce()
  3.06     42.83     1.79 54000000     0.00     0.00  Hphy::forward(double, double, double, double, double)
  2.86     44.49     1.67 52000000     0.00     0.00  Hphy::backward(double, double, double, double, double)
  2.58     46.00     1.51 54000000     0.00     0.00  Vphy::forward(double, double, double, double, double, double,
double)
  2.51     47.46     1.46 52000000     0.00     0.00  Uphy::backward(double, double, double, double, double, double,
double)
  2.50     48.91     1.46 54000000     0.00     0.00  Uphy::forward(double, double, double, double, double, double,
double)
  2.23     50.21     1.30       52     0.03     0.03  Yrazgrad_Hfil()
  2.10     51.44     1.23 52000000     0.00     0.00  Vphy::backward(double, double, double, double, double, double,
double)
  1.70     52.43     0.99 54000000     0.00     0.00  Hfil::forward(double, double, double)
  1.57     53.34     0.92 54000000     0.00     0.00  Ufil::forward(double, double, double)
  1.41     54.16     0.82 54000000     0.00     0.00  Vfil::forward(double, double, double)
  1.00     54.74     0.58       52     0.01     0.01  Yrazgrad_Vphy()
  1.00     55.32     0.58       52     0.01     0.01  Yrazgrad_Vfil()
  0.96     55.88     0.56       52     0.01     0.01  Yrazgrad_Hphy()
  0.94     56.43     0.55       52     0.01     0.01  Yrazgrad_Uphy()
  0.93     56.97     0.54       52     0.01     0.01  Yrazgrad_Ufil()
  0.63     57.34     0.37 52000000     0.00     0.00  Vfil::backward(double, double, double)
  0.56     57.66     0.33 52000000     0.00     0.00  Hfil::backward(double, double, double)
  0.42     57.91     0.25 52000000     0.00     0.00  Ufil::backward(double, double, double)
  0.22     58.04     0.13        1     0.13     0.13  Vphy::Vphy()
  0.07     58.08     0.04                             dd_
  0.05     58.11     0.03                             ctlcan_
  0.05     58.14     0.03                             m1qn3a_
  0.05     58.17     0.03                             sdot_
  0.03     58.19     0.02       51     0.00     0.00  Y3getstate_Hfil(float*)
  0.03     58.20     0.02        1     0.02     0.02  xvitgeo()
  0.02     58.21     0.01   520000     0.00     0.00  Ywishdiff_all(char*, int, int, int, int, int, double)
  0.02     58.22     0.01   520000     0.00     0.00  Ycostdiff(double, double)
  0.02     58.23     0.01   250000     0.00     0.00  select_io(int, char*, int, int, int, int, int, double*)
  0.02     58.24     0.01    10400     0.00     0.00  Ycobs2_list(Yst_fobs*)
  0.02     58.25     0.01    10000     0.00     0.00  Yobs_insert(Yst_obs*)
  0.02     58.26     0.01       52     0.00     0.00  Ysetwish_Hfil(double)
```

Figure V.6: Analyse par gprof du projet de test après optimisation

Le temps de traitement total passe de 108,52 secondes à 58,26 secondes (presque 50 % du temps initial). Les tests effectués sur les deux autres configurations donnent des valeurs similaires.

V.3.1.2 Analyse par Callgrind

L'analyse par Callgrind montre bien une réduction importante des défauts de cache dans le projet.

Procédure	Indicateur « cycle estimation »	Lectures en mémoire	Ecritures en mémoire	Instructions appelées	Défauts de cache de niveau 1	Défauts de cache de niveau 2
Ybackward_space_Soce (pourcentage)	42,56	39,54	42,61	42,32	44,58	46,57
Yforward_space_Soce (pourcentage)	17,26	18,64	19,38	18,24	16,95	0,1
Yvsmatt (pourcentage)	17,01	24,17	12,6	18,08	6,93	0,08
memset(pourcentage)	4,39	1,22	5,65	4,68	0	0
Total forward et backward modules (pourcentage)	11,45	12,33	13,63	10,4	17,59[5]	28,41
Total programme (valeur)	*8957705845*	*2954125532*	*1856951651*	*8395299275*	*8541977*	*4769868*

Tableau V.2: résultats de l'analyse par Callgrind suite à l'optimisation

Quelles différences peux-t-on remarquer par rapport à la configuration initiale ? Les résultats sont présentés dans le tableau V.2.

- L'indicateur « cycle estimation » représente 80% de la valeur présentée dans le tableau V.1.
- La quantité totale de défauts de cache a baissé d'un facteur 10 environ.
- La proportion entre les opérations sur la mémoire et les défauts de cache se sont rééquilibrées. De plus, la plupart des défauts de cache de niveau 2 proviennent de *Ybackward_space_Soce*, et ceux de niveau 1 se sont rapproché du nombre de lectures.
- Les défauts de cache provenant de la lecture de données représentent maintenant 99,9% de la totalité de ceux-ci.

V.3.1.3 Conclusion

Les résultats sont sans aucun doute concluants. Le temps d'exécution a été divisé par deux. Une optimisation des structures générées par Yao a donc été réalisée au niveau du générateur.

Toutefois, d'autres points sont améliorables comme la procédure *yvsmatt* qui, malgré la modification de la structure, prend toujours un quart du temps de traitement.

V.3.2 Retrait des éléments temporaires (accélération du lancement des forward/backward modules)

Tant pour le calcul de la passe avant d'un espace que pour celui de l'adjoint ou du tangent linéaire, un tableau est utilisé pour le passage des paramètres aux fonctions de

[5]Comme pour le test du programme d'origine, le ratio des méthodes backward est proche de zéro, en particulier pour le cache de niveau 2.

calculs des modules. Des instructions d'écritures dans ce tableau ont lieu avant chaque appel de la fonction concernée du module, comme on a pu le voir dans la figure V.4, présentant le code source d'une passe avant.

On remarque que l'ensemble des états est copié dans un tableau nommé *Yting*. Ce tableau est ensuite passé en paramètre valeur par valeur à la fonction forward du module concerné.

La copie des états dans la variable *Yting*, ajoutée au fait que les paramètres sont passés par valeur, provoque une double copie des éléments, ce qui peut être particulièrement coûteux. Le deuxième point peut être réglé par l'utilisation de certaines options d'optimisation du compilateur. En effet celles-ci peuvent déterminer automatiquement les fonctions à passer en « inline » par exemple, les utilisateurs n'ont cependant pas forcément l'habitude d'effectuer ce genre de manipulation automatiquement. De plus, les points négatifs liés à l'utilisation de cette technique restent dans ce cas inexistants. En effet, le code des modules est appelé une seule fois dans le code source du projet. Il est donc envisageable ici de modifier les passages de paramètre par valeur par des passages de paramètre par référence.

La copie dans le tableau *Yting* est plus compliquée à modifier. En effet, les tests aux limites présents dans la boucle rendent difficile le retrait du tableau. Il est envisageable d'éviter cette série de conditions par une technique de loop unrolling agissant sur les bords de l'espace, mais ceci provoquerait cependant une explosion du nombre de lignes de code.

A ce moment de l'analyse, j'ai pris connaissance du projet de d'optimisation et de parallélisation automatique du code. Pour cela, mes travaux à ce propos se sont arrêtés, et aucun test réel n'a donc été effectuée sur cette partie.

V.3.3 Utilisation de BLAS

Il est en général préférable d'utiliser des bibliothèques spécialisées au lieu de « réinventer la roue » en programmant des fonctions déjà existantes. Une amélioration peut être réalisée sur les produits matriciels présents dans le programme généré. En effet, ceux-ci sont codés dans une fonction. Comme nous l'avons vu dans la conclusion de la partie V.3.1, les calculs vecteur-matrice transposée nécessaires pour le calcul de la rétropropagation représentent de 20 à 25% du temps de calcul du programme Yao, ceux-ci étant réalisés par une procédure « maison ». Comme on a pu le voir dans la partie précédente, cette fonction est appelée 312.000.000 de fois (pour 5200 passes arrières) pour

la configuration d'espace de dimension 100. Une optimisation de ces traitements pourrait s'avérer particulièrement intéressante.

V.3.3.1 Libraries BLAS et dérivées

BLAS [32] (Basic Linear Algebra Subprograms) est un ensemble de routines fournissant les fonctions de base pour le calcul d'opérations sur les vecteurs et matrices. BLAS est scindée en trois « niveaux » :

- le niveau I pour les opérations concernant exclusivement les vecteurs
- le niveau II pour les opérations matrices-vecteurs
- le niveau III pour les opérations entre matrices

Les librairies d'algèbre linéaire s'appuient en général sur le standard « de facto » BLAS. De nombreux logiciels connus en emploient des implémentations de celle-ci comme par exemple MATLAB, SCILAB, etc. L'implémentation Netlib -considérée comme étant celle de référence- ne fournit pas les meilleurs performances. Il existe plusieurs implémentations de BLAS optimisées (ATLAS, GotoBLAS) suivant le contrat d'interface de Netlib.

BLAS est souvent associée à LAPACK [37]. Il s'agit également d'une bibliothèque standard de calcul pour l'algèbre linéaire. Elle s'appuie sur BLAS et permet de faire des calculs plus complexes comme par exemple des déterminations de valeurs propres de matrices ou des résolution de problèmes d'algèbre linéaire. Dans l'état actuel du logiciel YAO, nous n'aurons pas besoin de LAPACK. En effet, les opérations réalisées par Yao pour le tangent linéaire et la rétropropagation sont des multiplications sur des vecteurs ou des matrices. Toutefois, il faut garder à l'esprit que BLAS et LAPACK peuvent être éventuellement utiles pour des évolutions futures du logiciel ou pour fournir une boite à outils de manipulation de matrices pour les utilisateurs de YAO.

Il est à noter que les bibliothèques BLAS et LAPACK ne proposent pas de fonctionnalité distribuée. Pour cela, des alternatives basées directement sur BLAS comme SCALAPACK [38] et PBLAS [39] existent. Toutefois, l'intérêt de ce type de bibliothèques reste cependant faible pour les projets Yao. En effet, le principe de Yao étant de décomposer un modèle en de nombreux modules de taille limitée, les calculs matriciels réalisés par Yao lors de la rétropropagation et du calcul du tangent linéaire se font en général sur des matrices et vecteurs de taille très faible.

L'étude comparative proposée par Mouton [34] pour l'outil d'assimilation Verdandi [41] développée par l'INRIA présente une très bonne synthèse des bibliothèques existantes en C++. Il est à noter que les fonctions des librairies BLAS sont le plus souvent écrites en Fortran ou en langage C. Il existe des adaptateurs C (comme CBLAS [36]) et C++ (comme par exemple Seldon [42] - l'outil retenu par l'INRIA pour Verdandi) permettant une utilisation par un projet non codé en Fortran. Suite à une analyse de ce document associée à une recherche sur Internet, voici une liste non exhaustive de bibliothèques respectant les standards BLAS/LAPACK. Celle-ci est décrite par le tableau V.3.

	Netlib	**Eigen**	**GotoBLAS**	**ATLAS**	**Intel MKL**	**ACML**
Langage	Fortran	C++	Fortran	Fortran/C	Fortran/C	Fortran/C
Licence	Libre	LGPL	BSD	BSD	Non libre Payant	Non libre Gratuit
Besoin wrapper C++	Oui	Non	Oui	Oui	Oui	Oui
BLAS	Oui	Oui	Oui	Oui	Oui	Oui
LAPACK	Oui	Incomplet	Non	Incomplet	Oui	Oui
Vectorisation (SIMD)	Non	Oui (SSE, Altivec, ARM Neon)	Oui (sur plateformes supportées)	Non	Oui (SSE)	Oui (SSE)
Optimisation cache	Non	Oui	Oui	Oui	Oui	Oui
Multithread	Non	Oui (OpenMP)	Oui (OpenMP)	Non	Oui	Oui
Support matrices taille fixe	Non	Oui	Non	Non	Non	Non
Plateformes supportées	Toutes avec compilateur GCC ISO	Toutes avec compilateur GCC ISO	x86, x86_64, IA64, Power, SPARC, Alpha	Toutes avec compilateur GCC ISO	Intel	AMD
Performances	Mauvaises	Bonnes	Bonnes	Moyennes	Bonnes	Bonnes

Tableau V.3: Tableau comparatif entre quelques bibliothèques de calculs matriciels. Les informations ont été obtenues sur les sites respectifs.

Les données ont été récupérées sur les sites respectifs. L'appréciation des performances reste évidemment subjective, vis à vis de l'ensemble des tests visualisés.

Il est préférable d'utiliser un wrapper C++ pour une utilisation dans du code du même langage. En effet, le source généré par Yao est en C++, et l'utilisation de classes dédiées simplifiera le travail des utilisateurs et du développement du générateur.

Yao pourrait tirer parti des éventuelles bibliothèques BLAS optimisées installées sur les postes. Cela ajoute plus de portabilité au logiciel, mais complexifie le déploiement.

Dans le cas où l'on préférerait avoir un adaptateur C++ permettant de profiter des implémentations de BLAS présentes sur la machine, la bibliothèque Seldon semble la plus appropriée.

V.3.3.2 Tests effectués

J'ai effectué des tests avec le wrapper C++ Seldon, en utilisant l'implémentation par défaut de BLAS présente sur la machine de test (Netlib compilée pour processeur 64 bits), ainsi qu'avec ATLAS et GotoBLAS. Il m'a également été nécessaire d'utiliser le wrapper C CBLAS pour le test de l'implémentation Netlib. Les interfaces C étaient fournies avec les deux autres bibliothèques.

Le projet de test consiste en l'utilisation de la fonction *yvsmatt* citée dans les parties précédentes appliquée à un tableau statique, de la même manière que dans la boucle interne de traitement de Yao.

Un point à souligner est que selon le nombre de sorties du module concerné, la matrice jacobienne peut se retrouver être un vecteur dans le cas où le module aurait une seule sortie. Dans ce type de cas de figure, l'opération à réaliser est une multiplication de vecteur par un scalaire. Les librairies BLAS fournissent des opérations séparées pour les multiplications vecteur-matrice et scalaire-vecteur, les deux étant souvent optimisées d'une manière différente. Les performances des deux fonctions ont été testées.

Les performances des différentes implémentations testées (Netlib, GotoBLAS, ATLAS), sont comparables. Les tests effectués sur la multiplication vecteur-matrice transposée donnent une amélioration de 10% sur cette fonction. Le gain appliqué à l'ensemble du programme résulterait en une amélioration de 2.5% pour un projet constitué seulement de modules avec une seule sortie. Pour le cas des multiplications scalaire-vecteur, l'amélioration se ramène à 20% par instruction en moyenne.

Aucune implémentation n'a été réalisée au niveau du générateur, cependant ceci donne des indications intéressantes pour le futur développement de Yao.

V.4 Perspectives

Actuellement, YAO n'utilise pas le jeux d'instructions SSE des processeurs x86. Cependant, ceci n'est pas forcément évident au vu du fait qu'il n'existe pas d'instructions standard en C/C++ concernant la vectorisation. Il existe une fonctionnalité de vectorisation automatique d'un programme à l'intérieur du compilateur gcc, mais, à ce jour, celle-ci n'est pas encore totalement au point. J'ai effectué des tests en ajoutant l'option au compilateur mais ceci n'a donné aucune amélioration de performance. Il est à noter que le sujet est encore actuellement à l'état de recherche. La vectorisation des instructions peut être toutefois une piste dans l'augmentation des performances de YAO.

Actuellement, l'ensemble des états de sortie et des gradients sont stockés, ce pour l'ensemble des modules, pour chaque espace et trajectoire. Ce besoin est dicté par la nécessité de garder l'ensemble des états et gradients calculés. Des améliorations sont possibles concernant ce point, via la mise en place d'une technique de checkpointing. Le lecteur pourra se référer à la thèse de Luigi Nardi [13] pour des explications plus précises concernant le checkpointing.

Il n'est également pas obligatoire de garder l'ensemble des gradients, à fortiori si le module est linéaire. En effet, dans le cas d'une relation linéaire entre les sorties et les entrées du modules, les valeurs du jacobien restent toujours identiques quelque soit la valeur des entrées. Ceci est envisageable de manière simple en ajoutant au niveau du fichier de description une instruction indiquant

Un autre outil est actuellement étudié dans le cadre d'une coopération avec le LIP6. Il s'agit du logiciel Pluto [41], permettant d'effectuer une parallélisation automatique du code source. Cet outil s'appuie sur la génération d'un modèle polyédrique basé sur les boucles d'un programme et les dépendances entre les variables utilisées dans celles-ci. Le principe de cette technique est d'effectuer des transformations linéaires du polyèdre obtenu de manière à obtenir des dépendances linéaires entre les zones mémoires. Ceci permet théoriquement d'augmenter la quantité de code source parallélisable, en réduisant parcourant les boucles de manière à limiter les dépendances croisées.

VI Conclusion

Durant cette année de stage, j'ai apporté de nombreuses évolutions importantes ont au logiciel Yao.

L'assimilation quasi-statique a montré son efficacité dans l'utilisation d'un modèle de complexité faible. L'avantage de cet algorithme est qu'il peut être très facilement appliqué à n'importe quelle algorithme existant et possède un nombre très faible de paramètres. Ceci garantit son utilisation et l'obtention simple de meilleurs résultats pour les modèles complexes.

L'implémentation de l'algorithme dual s'est montrée plus efficace que l'algorithme normale dans le cas d'un modèle imparfait. Il reste cependant à le tester de manière plus approfondie sur des modèles réels, ce qui sera le rôle de personnes plus spécialisées dans l'assimilation.

Le refactoring effectué sur les classes de Yao reste partiel, cependant il pose les bases d'une restructuration et d'une grande simplification des projets générés. La simplificiation effectuée fournit déjà une plus grande possibilité de personnalisation aux générateurs de projet, par le système des patrons, ainsi que les classes de lancement de modèle. Les classes d'observation implémentées actuellement seulement sur l'algorithme dual fournissent une première implémentation avant la généralisation

L'optimisation des boucles réalisée a permis un gain de temps précieux pour l'ensemble des projets d'assimilation s'appuyant sur Yao. L'utilisation future d'une bibliothèque de matrices permettra également d'obtenir des résultats rapidement pour les futures évolutions de Yao.

La mise en place de la dernière version du minimiseur m1qn3 a permis le passage de Yao en licence CECILL, et sa distribution.

Durant ce stage, j'ai pu apprendre à connaître un monde extrêmement intéressant, celui de la recherche. Mes connaissances limitées en mathématiques ont rendu certaines des tâches difficiles, mais ceci m'a permis d'affermir ma culture générale, et m'a donné envie de continuer dans ce domaine.

Bibliographie

[1] DAGET N., 2007. *Revue des méthodes d'assimilation*. Rapport technique, CERFACS, 76 p.

[2] « Automatic differentiation - Wikipedia, the free encyclopedia ». Disponible sur : < http://en.wikipedia.org/wiki/Automatic_differentiation > (consulté le 15 janvier 2012)

[3] « Web Server of the TROPICS team ». Disponible sur : < http://www-sop.inria.fr/tropics/ > (consulté le 22 juin 2011)

[4] « TAF ». Disponible sur : < http://www.fastopt.de/products/taf/taf.html > (consulté le 15 février 2011)

[5] « TAC++ ». Disponible sur : < http://www.fastopt.de/products/tac/tac.html > (consulté le 15 février 2011)

[6] « OpenAD ». Disponible sur : < http://www.mcs.anl.gov/OpenAD/ > (consulté le 15 février 2011)

[7] « The OpenPALM dynamic parallel coupler ». Disponible sur : < http://www.cerfacs.fr/globc/PALM_WEB/ > (consulté le 15 février 2011)

[8] « OASIS - vERC ». Disponible sur : < https://verc.enes.org/models/software-tools/oasis > (consulté le 15 février 2011)

[9] « www.Autodiff.org - AD Tools for ALL ». Disponible sur : < http://www.autodiff.org/?module=Tools&language=ALL > (consulté le 15 février 2011)

[10] CHAPELLE D. et al., 2010. *Revue des bibliothèques pour l'assimilation de données et exemple de la bibliothèque Verdandi*. Rapport technique, INRIA, 23 p.

[11] « UCAR The Data Assimilation Research Testbed -- DART ». Disponible sur : < http://www.image.ucar.edu/DAReS/DART/index.php > (consulté le 15 février 2011)

[12] « CeCILL ». Disponible sur : < http://www.cecill.info/index.fr.html > (consulté le 22 août 2011)

[13] NARDI L., 2011. *Formalisation et automatisation de YAO, générateur de code pour l'assimilation variationnelle de données*. Thèse de docteur, Conservatoire National des Arts et Métiers Paris, 147 p.

[14] BERNARD C., 2010. *Aide à la gestion des parcours de l'espace sous le logiciel Yao*. Mémoire d'ingénieur, Conservatoire National des Arts et Métiers Paris, 88 p.

[15] « M1QN3 ». Disponible sur : < http://www-roc.inria.fr/~gilbert/modulopt/optimization-routines/m1qn3/m1qn3.html > (consulté le 10 décembre 2010)
[16] KANE A., 2010. *Assimilation de données in situ et satellitaires dans le modèle de biogéochimie marine PISCES*. Thèse de docteur, Université de Versailles Saint Quentin en Yvelines, 165 p.
[17] THIRIA S., BADRAN F., SORROR C., 2006. *YAO : Un logiciel pour les modèles numériques et l'assimilation de données*. Rapport de recherche, LOCEAN Paris, 63 p.
[18] BRAJARD J. *Méthodologie neuronale pour l'inversion des signaux satellitaires de couleur de l'océan. Traitement des aérosols absorbants et restitution de la concentration en chrolorophylle-a*. Thèse de docteur, Université Pierre et Marie Curie Paris VI, 203 p.
[19] « NEMO Home Page - NEMO Website ». Disponible sur : < http://www.nemo-ocean.eu/ > (consulté le 10 janvier 2012). http://www.nemo-ocean.eu/
[20] « OpenMP.org ». Disponible sur : < http://openmp.org/wp/ > (consulté le 20 octobre 2011)
[21] « M2QN1 ». Disponible sur : < https://who.rocq.inria.fr/Jean-Charles.Gilbert/modulopt/optimization-routines/m2qn1/m2qn1.html > (consulté le 20 novembre 2011)
[22] « ANTLR Parser Generator ». Disponible sur : < http://www.antlr.org/ > (consulté le 20 November 2010)
[23] « LL(*) grammar analysis ». Disponible sur : < http://www.antlr.org/wiki/display/~admin/LL(*)+grammar+analysis > (consulté le 20 novembre 2010)
[24] « Boost C++ Libraries ». [s.l.] : [s.n.], [s.d.]. Disponible sur : < http://www.boost.org/ > (consulté le 20 janvier 2011)
[25] « The Boost Graph Library - Boost 1.47.0 ». Disponible sur : < http://www.boost.org/doc/libs/1_47_0/libs/graph/doc/index.html > (consulté le 20 janvier 2011)
[26] « AutoGen: The Automated Text and Program Generation Tool ». Disponible sur : < http://www.gnu.org/s/autogen/ > (consulté le 27 février 2011)
[27] PIRES C., VAUTARD R., TALAGRAND O., 1996. On extending the limits of variational assimilation in non linear chaotic systems. *Tellus A*. 48, 96-121.

[28] FERRON B., 2011. A 4D-variational approach applied to an eddy-permitting North Atlantic configuration: Synthetic and real data assimilation of altimeter observations. *Ocean Modelling*. 39, n°3-4, 370–385.
[29] ROULSTON M. S., 1998. Data assimilation in chaotic systems. *Astrophysical and geophysical flows as dynamical systems: 1998 Summer Study Program in Geophysical Fluid Dynamics*. Rapport technique, DTIC, 12 p.
[30] CHELLAPA S., FRANCHETTI F., 2007. *How To Write Fast Numerical Code: A Small Introduction,* 68 p.
[31] « Locality of reference - Wikipedia, the free encyclopedia ». Disponible sur : < http://en.wikipedia.org/wiki/Locality_of_reference > (consulté le 2 novembre 2011)
[32] GRAHAM S., KESSLER P., MCKUSICK M., 1982. gprof: a Call Graph Execution Profiler. *Proceedings of the 1982 SIGPLAN symposium on Compiler construction.* 17, 6, 120-126.
[33] « Valgrind ». Disponible sur : < http://valgrind.org/docs/manual/manual.html > (consulté le 10 février 2011)
[34] NETHERCOTE N., 2004. *Dynamic Binary Analysis and Instrumentation*. Thèse de doctorat. Université de Cambridge, 190 p.
[35] « KCacheGrind ». Disponible sur :
< http://kcachegrind.sourceforge.net/html/Home.html > (consulté le 10 décembre 2010)
[36] « BLAS ». Disponible sur : < http://www.netlib.org/blas/index.html > (consulté le 28 mars 2011)
[37] « LAPACK - Linear Algebra PACKage ». Disponible sur :
< http://www.netlib.org/lapack/ > (consulté le 29 mars 2011)
[38] « ScaLAPACK - Scalable Linear Algrabra PACKage ». Disponible sur :
< http://www.netlib.org/scalapack/ > (consulté le 29 mars 2011)
[39] « PBLAS Home Page ». Disponible sur :
< http://www.netlib.org/scalapack/pblas_qref.html > (consulté le 30 mars 2011)
[40] MOUTON C., 2009. *A study of the existing linear algebra libraries that you can use from C++*. Rapport Technique. INRIA, 41 p.
[41] « Verdandi, data assimilation library ». Disponible sur :
< http://verdandi.gforge.inria.fr/ > (consulté le 5 avril 2011)
[42] « Seldon C++library - linear algebra ». Disponible sur :
< http://seldon.sourceforge.net/ > (consulté le 11 avril 2011)

Liste des figures

Liste des tableaux

Mémoire d'Ingénieur C.N.A.M., Paris 2012

RESUME

Le logiciel Yao est un générateur de code source dont le but est d'effectuer des sessions d'assimilation variationnelle. Une de ses fonctionnalités est l'aide à l'utilisateur à la mise au point de simulations numériques en s'appuyant sur la notion de graphe modulaire pour la représentation d'un modèle. Il fournit un environnement de développement permettant à l'utilisateur d'effectuer des sessions d'assimilation à l'aide de l'algorithme standard et de l'algorithme incrémental. Il fournit également des fonctions de test et de contrôle. Ce mémoire présente les évolutions ayant été apportées à Yao durant le stage de Guillaume Rosinosky. Une refactorisation du code source des projets générés a été effectuée dans le but de simplifier la maintenance et la lisibilité du code, ce à l'aide de représentations objet des différents concepts en présence. Deux nouveaux algorithmes ont été ajoutés, et testés. L'implémentation de l'algorithme quasi-statique permet d'assimiler des modèles de nature plus complexe. L'algorithme d'assimilation à contrainte faible permet à l'utilisateur de tenir compte des erreurs modèle par le passage en forme duale des équations de l'assimilation variationnelle. En plus de ces modifications majeures, une optimisation des performances des programmes générés a été également réalisée.

Mots clés : assimilation variationnelle, optimisation, assimilation variationnelle à contrainte faible, Yao, assimilation variationnelle quasi-statique

---

SUMMARY

The Yao software is a source code generator which aims to perform variational assimilation runs. One of its fonctionnalities is to help the user to conceive numeric simulations by using the notion of a modular graph for model representation. It gives an integrated development environment which provides the user with the ability to make standard and incremental variational assimilation sessions. Furthermore, it supplies a number of test and control functions. In this dissertation, we present the evolutions developed in the Yao software during the internship of Guillaume Rosinosky. A refactoring of the source code of the generated project has been done in order to simplify the maintenance and the readability of the source code, by using oriented object representations of the internal concepts of Yao. Two new algorithms have been incorporated and tested. The implementation of the quasi-static algorithm open the possibility to assimilate more complex models. The variational assimilation in weak constraint algorithm allows to take into consideration the model errors by switching in dual form the variational assimilation equations. In addition to those major modifications, general optimisation of the performances have been applied.

Key words : variational assimilation, optimisation, variational assimilation in weak constraint, Yao, quasi-static variational assimilation