Cahiers des algorithmes de Mini-ker

Al1, Août 2006

Versions 1.02.xx

Ce manuel est une description de l'environnement développé pour l'utilisation pratique de Mini_ker dans ses versions 1.02[1]*. Il s'agit de notes permettant de comprendre le code Mortran de Mini_ker* [2]*, et de pouvoir le développer dans le même esprit.*

Le lecteur est renvoyé aux différents documents concernant les bases théoriques du TEF, de l'adjoint et des sensibilités, cités dans les pages **ZOOM**[3]*, ainsi que ce qui concerne le filtre de Kalman.*

Le découpage des séquences de calcul du programme **principal** *de Mini_ker suit une logique de présentation en même temps que l'ordre des instructions, excluant cependant des options supplémentaires. A la lecture, il faut ainsi parcourir plusieurs fois un exemple de* **exo_o.tmp** *en sautant des passages du code pour suivre les explications du Cahier.*

Enfin, la rédaction de ce cahier se faisant en parallèle avec la programmation de la version **102**, on notera le passage progressif de routiines de la **mathlib** ancienne ou nouvelle convention (ex. matmul -¿ omatmlt) LaPack.

[1]à quelques détails près, il suffit aussi à comprendre les principes des versions précédentes, qui ne bénéficient pas de ce type de manuel.

[2]il est ainsi conseillé d'avoir sous la main un ../cfs/exo_o.tmp pour lire ces notes.

[3]*http ://www.lmd.jussieu.fr/Zoom*

Principes et structure du Mini_ker

Introduction générale aux choix de programmation.

– définition de langages dédiés dès qu'ils se dégagent

– principe d'unicité (ou de non-duplication) des calculs ;

L'écriture de langages pour tenter une remontée de la couche algorithmique est une méthode systématique de la collaboration ZOOM. L'utilisation du précompilateur Mortran[1] offre les facilités nécessaires à l'écriture symbolique des descriptions de modèles et des diverses options de calcul. Remarquons cependant que par rapport à ZOOM, cette systématique en est restée à un niveau modeste dans Mini_ker, du fait de la simplicité des algorithmes et de la gestion des matrices, qui sont rangées en un seul bloc par matrice[2].

Le code Fortran généré utilise des séquences de calcul répétées dans des contextes différents, pour le calcul de la trajectoire, des sensibilités et de l'adjoint. On a donc découpé ce code final en séquences (des paquets d'instructions Mortran ou Fortran) ; ces séquences sont gérées par CMZ[3]

[1]SLAC, CERN et RAMSÈS.

[2]un seul niveau d'emboitement est envisagé pour la version 200.

[3]CERN Code Management using Zebra.

1 Résumé des calculs

1.1 Calculs de trajectoire

Le principe de Mini_ker est celui du TEF-ZOOM en ne conservant qu'une cellule et qu'un transfert (avec, ainsi, suppression de l'emboîtement des familles[1]).

Un système est ainsi défini par

$$\begin{aligned} \partial_t \eta(t) &= g(\eta(t), \varphi(t), t) \\ \varphi(t) &= f(\eta(t), \varphi(t), t) \end{aligned} \tag{1}$$

duquel on dérive le système d'avance le long de l'espace tangent local :

$$\begin{bmatrix} \partial_t \delta\eta \\ 0 \end{bmatrix} = \begin{bmatrix} H & B_b \\ C^\dagger & D \end{bmatrix} \begin{bmatrix} \delta\eta \\ \delta\varphi \end{bmatrix} + \begin{bmatrix} \Gamma \\ \Omega \end{bmatrix} \tag{2}$$

dans ce système, les $\delta x(\tau), \tau \in [t, t+\delta t]$ sont des vecteurs évoluant dans le tube osculant la trajectoire dans l'espace tangent local, valable de t à $t+\delta t$. Les matrices sont les Jacobiennes de (1) :

$$\begin{aligned} A = \partial_\eta g \quad &; \quad B_b = \partial_\varphi g; \\ C^\dagger = \partial_\eta f \quad &; \quad D = \partial_\varphi f \end{aligned} \tag{3}$$

Le vecteur $\Gamma = \delta t(g((t)) + \frac{\delta t}{2}\partial_t g$, et Ω est en général nul sauf cas particulier (source, transfert intégré) et est ici conservé par souci de généralité de la forme de ce système – cf calcul des sensibilités.

Après application du schéma de discrétisation au deuxième ordre en temps du TEF, on aboutit au système algébrique suivant

$$\begin{bmatrix} A & B \\ -C^\dagger & I-D \end{bmatrix} \begin{bmatrix} \Delta\eta \\ \Delta\varphi \end{bmatrix} = \begin{bmatrix} \Gamma \\ \Omega \end{bmatrix} \tag{4}$$

avec les nouvelles matrices dépendant de δt et des Jacobiennes : $A = I - \frac{\delta t}{2}H$ et $B = -\frac{\delta t}{2}B_b$.

La résolution suit les principes du TEF en appliquant une première étape d'élimination des variables $\delta\eta$, qui donne le système dit "de couplage" (entre composantes du transfert) :

$$[\ I - D + C^\dagger A^{-1} B\ ]\ \delta\varphi = \Omega + C^\dagger A^{-1}\Gamma \tag{5}$$

C'est la matrice $\mathcal{C} = [\ I - D + C^\dagger A^{-1} B\ ]$ qui est dite " de couplage", et le vecteur $\delta\varphi_{ins} = \Omega + C^\dagger A^{-1}\Gamma$ représente l'évolution insensible du transfert (unique) de Mini_ker.

Après résolution du système linéaire (5), on a $\delta\varphi$, et il reste à calculer l'avance d'état : $\delta\eta = \Gamma + A^{-1}B\delta\varphi$, ce qui finit de déterminer l'avance du système en dt. L'état suivant $\eta(t+\delta t) = \eta(t) + \delta\eta$ étant connu, on détermine le nouveau vecteur des transferts par la résolution du système implicite $\varphi(t) = f(\eta(t), \varphi(t), t)$, qui doit donner une solution peu éloignée de celle-ci

$$\varphi(t+\delta t) = \varphi(t) + \delta\varphi \quad \sim \quad \textit{solution de } \varphi' = f(\eta(t+\delta t), \varphi')$$

1.2 Calculs de sensibilité

Le système déterminant les sensibilités aux conditions initiales est

$$\begin{bmatrix} \partial_t \Phi \\ \Psi \end{bmatrix} = \begin{bmatrix} H & B_b \\ C^\dagger & D \end{bmatrix} \begin{bmatrix} \Phi \\ \Psi \end{bmatrix} (t, \tau) \tag{6}$$

[1] du moins jusqu'à la version 2.00.

et s'intègre ainsi comme le système d'avance (2) – avec la même matrice de couplage. Les inconnues sont à présent les matrices Φ etΨ, qui étendent alors les vecteurs $\delta\eta, \delta\varphi$, avance $\Phi(0,0) = I$ matrice unité.

On profite de cette propriété pour résoudre ce système en même temps que l'avance de trajectoire. Les différences ne portent que sur les termes sources, c'est-à-dire les vecteurs étendant Γ, Ω.

L'application dans Mini_ker est alors très simple :
- les jacobiennes sont calculées par dérivation formelle des modèles formulés au début du code (**principal**) ;
- on passe des Jacobiennes au calcul des matrices de l'avance en prenant en compte l'incrément temporel ;
- l'appel au sous-programme **ker** effectue l'élimination des cellules (5), l'appel à LaPack (SGESV) pour la résolution de ce dernier système linéaire, mais complété par un certain nombre choisi de colonnes de Φ, Ψ.
- L'avance de η, Φ est alors effectuée dans **principal** au retour de **ker** ;
- il ne reste plus qu'à résoudre l'avance implicite de φ.
- Quand ceci est fait, la matrice D est déterminée, $C^\dagger$ peut être calculée, ce qui suffit à déterminer Ψ.

En résumé, on intègre dans le même mouvement un ensemble de systèmes possédant les mêmes Jacobiennes, seule la trajectoire nécessitant la résolution d'un système implicite. Les différents systèmes ne diffèrent que par leurs vecteurs connus et leurs C.I.

En particulier, le système des sensibilités (cf cahier Effluents) mène à une variété de sources et donc, vecteurs de type Γ, Ω, comme nous l'expliciterons en temps utiles.

1.3 Calculs dans la version 1.02

La version Mini_ker 102 intègre en avant, en plus de la trajectoire et des sensibilités aux variables du TEF :
- un calcul de l'avance d'un type de transfert nouveau, dit "d'observables" (ou "probes") : vecteur μ ;
- le calcul des sensibilités des variables du TEF à une liste de paramètres – définis par leurs symboles Fortran dans la formulation des modèles (formant un vecteur π ordonné par la liste) ;

Ce système a pour équation :

$$\begin{bmatrix} \partial_t \delta\varpi \\ \partial_t \delta\eta \\ \delta\varphi \\ \delta\mu \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ H_\varpi & H & B_b & 0 \\ C^\dagger_\varpi & C^\dagger & D & 0 \\ c_1 & c_2 & d_{\varphi\mu} & 0 \end{bmatrix} \begin{bmatrix} d\varpi \\ \delta\eta \\ \delta\varphi \\ \delta\mu \end{bmatrix} + \begin{bmatrix} 0 \\ \mathit{termes} \\ \mathit{sources} \\ 0 \end{bmatrix} \tag{7}$$

dans lequel on peut faire les remarques suivantes :
- les paramètres sont des cellules – puisqu'il faut définir des C.I., d'équation $\partial_t \varpi = 0$;
- les variables sont insensibles aux obs par définition ; attention, dans le cas du filtrage optimal de Kalman, il pourra y avoir bouclage par une matrice de gain K ;
- on a *a priori* exclu tout couplage instantané (par une matrice D_μ) entre les obs, ni envisagé de sources ; ceci peut évidemment être changé ;
- le système a la même matrice de couplage que (1) ;

La conséquence sur le calcul reste standard : on a d'une part à faire avancer la trajectoire, de l'autre à calculer des vecteurs connus supplémentaires ainsi que des systèmes linéaires explicites supplémentaires. Pour la raison que le compilateur Fortran n'accepte pas les dimensions de tableau nulles, on étend la dimension des matrices $H, C^\dagger$ et D, l'objectif restant de ne pas stocker de blocs-matrices nuls.

Nous allons détailler les calculs à effectuer. Le système algébrique à intégrer est à présent :

$$\begin{bmatrix} A & B & 0 \\ C^\dagger & I-D & 0 \\ -c_2 & -d_{\varphi\mu} & I \end{bmatrix} \begin{bmatrix} \delta\eta \\ \delta\varphi \\ \delta\mu \end{bmatrix} = \begin{bmatrix} \Gamma \\ \Omega \\ 0 \end{bmatrix} \tag{8}$$

1.3.1 Calculs de la trajectoire

Standards, puisque ϖ n'avance pas et que les variables standards sont insensibles aux obs. On avance les obs par :

$$\Psi_\mu = c_2\delta\eta + d_{\varphi\mu}\delta\varphi \tag{9}$$

et ceci dès que la détermination implicite de φ a été effectuée et après calcul des jacobiennes.

1.3.2 Calculs des sensibilités

En remarquant que la sensibilité de ϖ à ϖ reste égale à I, écrivons-en le système complet des sensibilités correspondant au SLTC (7) avant de le commenter :

$$\begin{bmatrix} \partial_t[I\,,\,0] \\ \partial_t[\Phi_\varpi, \Phi] \\ \Psi_\varpi, \Psi \\ \Psi_{\varpi\mu}, \Psi_\mu \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ H_\varpi & H & B_b & 0 \\ C^\dagger_\varpi & C^\dagger & D & 0 \\ c_1 & c_2 & d_{\varphi\mu} & 0 \end{bmatrix} \begin{bmatrix} I, 0 \\ \Phi_\varpi, \Phi \\ \Psi_\varpi, \Psi \\ \Psi_{\varpi\mu}, \Psi_\mu \end{bmatrix} + \begin{bmatrix} 0,0 \\ termes \\ sources \\ 0,0 \end{bmatrix} \tag{10}$$

dans lequel on a mis à part les colonnes relatives aux paramètres (ϖ).

On peut donc oublier la première ligne du système. La sous-matrice H_ϖ n'intervient que dans la partie explicite ($\Gamma = g(t)\delta t$) de ce système linéaire. On a donc simplement à résoudre le système étendu suivant :

$$\begin{bmatrix} A & B & 0 \\ -C^\dagger & I-D & 0 \\ -c_2 & -d_{\varphi\mu} & 0 \end{bmatrix} \begin{bmatrix} \delta\Phi \\ \delta\Psi \\ \delta\Psi_\mu \end{bmatrix} = \begin{bmatrix} \Gamma_\Phi \\ \Omega_\Psi \\ 0 \end{bmatrix} \tag{11}$$

commentaires :

- la partie explicite de la première ligne est $\Gamma_\Phi = \delta t[H\Phi + B_b\Psi]+$ sources et pour les colonnes relatives à ϖ : $\Gamma_{\varpi\Phi} = \delta t[H_\varpi + H\Phi_\varpi + B_b\Psi_\varpi]$;
- la détermination de Ψ se fait par $(I-D)\Psi = C^\dagger\Phi+$ sources ;
- celle de Ψ_ϖ par $(I-D)\Psi_\varpi = c1 + c_2\Phi_\varpi$;
- le terme supplémentaire est donné par $\Psi_\mu = c_2\Phi + d_{\varphi\mu}\Psi$ et $\Psi_{\varpi\mu} = c_1 + c_2\Phi_\varpi + d_{\varphi\mu}\Psi_\varpi$ (cf 10), et ainsi la dernière ligne de (11) n'est pas utilisée (on ne calcule pas les $d\Psi_\mu$) ;
- les termes sources des vecteurs Γ_Φ er Ω_Ψ sont à définir colonne par colonne en fonction du type de sensibilité choisi ;
- dans le cas des sensibilités à n_I C.I., c'est n_I colonnes de $\Phi(0,0)$ qui sont mises à un coefficient 1 ;
- le cas des sensibilités à une perturbation de n_f composantes de transfert mène à au moins deux cas standards déjà envisagés (cf Effluents), c'est-à-dire un step avec source unité (qui ira dans une colonne de Ω), et celui d'un pulse – qui consistera à remplir une colonne de Γ par une colonne de $(I-D)^{-1}B$ à l'instant du pulse ; on pourra étendre à des perturbations supplémentaires (rampe, etc) ;

Par ailleurs, des raisons pratiques – telle l'existence d'un transitoire initial – il est souhaitable de pouvoir choisir un premier instant d'appplication des sources (**... at 100 ;**). Même en ce qui concerne la sensibilité aux C.I., on peut être amené à se désintéresser du transitoire. Par cohérence, on donne la même possibilité aux sensibilités paramétriques.

Enfin, on offre également la possibilité de ré-initialiser ces sensibilités à intervalles réguliers par (**... at 12 every 100 ;**). Cette option automatique peut toujours être effacée par une gestion directe des flags de contrôles (variables logiques **Zo_p(.)**) dans la séquence **ZSteer**).

2 Les variables et symboles, rangements

On fait ici le lien entre le formulaire et les tableaux Fortran du calcul, avec la spécificité de Mini_ker qui utilise un code symbolique dont certains symboles seront remplacés par des expressions ou symboles différents en fonction des macros générées par le **ZINIT**. Pour simplifier, on reporte à la section de l'adjoint la description des éléments dépendants de la sensibilité de la fonction de coût aux variables (adjointes), à la commande et aux paramètres.

On commence par un tableau des variables de type état et transferts – incluant donc les extensions :

nom	dim 1	dim 2	*signification*	data file
eta	np	1	vecteur d'état	res.data
dot_eta	np	1	vel d'état	no data
dneta	np	1	avance d'état	dres.data
Phi_t	np	nxp+1	matrice des eta-sensibilités aux variables	sens.data
dPhi_t	np	nxp+1	matrice des eta-sensibilités aux variables	no data
Phi_p	np	lp+1	matrice des eta-sensibilités aux paramètres	sensp.data
dPhi_p	np	lp+1	matrice des eta-sensibilités aux paramètres	no data
ff	mp	1	vecteur de transfert	tr.data
ff(mp+1)	mobs	1	vecteur des observables	obs.data
Psi_t	mp	nxp+1	matrice des ff-sensibilités aux variables	sigma.data
Psi_p	mp	lp+1	matrice des ff-sensibilités aux paramètres	sigmap.data
Pso_t	mobs	nxp+1	matrice des mu-sensibilités aux variables	sigmo.data
Pso_p	mobs	lp+1	matrice des mu-sensibilités aux paramètres	sigmop.data

On remarque que les dimensions +1 ne sont là que pour éviter une erreur de compilation avec une déclaration de dimension nulle. La solution implémentée est de déclarer que la colonne 0 d'un tableau est "equivalent" à la dernière du précédent :

```
real Phi_t(np,0:nxp),Psi_t(mp,0:nxp);
real Phi_p(np,0:nxp),Psi_p(mp,0:nxp);
equivalence (Phi_t(1,0),eta(1)),(Psi_t(1,0),ff(1));
equivalence (Phi_p(1,0),Phi_t(1,nxp)),(Psi_p(1,0),Psi_t(1,nxp));
```

Un ensemble de symboles Mortran sont dans les instructions de calcul des matrices et vecteurs de **principal**, qui seront remplacées par des expressions ou des symboles différents selon les directives générées à partir du **ZINIT**. Les voici :

nom	*signification*	origine du remplacement	selflag
deta_tef[i]	$g_i(\eta, \varphi)$	fun : de set_eta < ... >;	
deta_tef[i]	$g_i(\eta, \varphi)$	fun : de set_dwn_eta < ..., >;	Grid1D
nde_eta_fun_[.i](inode)	$g_{i+nlo}(\eta, \varphi)$	fun : de set_node_eta < ..., >;	Grid1D
deta_tef(up[.i])	$g_{i+o}(\eta, \varphi)$	fun : de set_up_eta < ..., >;	Grid1D
(/eta(np+[.j]))	composante de ϖ	free_parameters : liste	tjrs
dphi_tef[i]	$f_i(\eta, \varphi)$	set_phi < ...,fun :... ; ...>;	
phi_tef[i]	$f_i(\eta, \varphi)$	fun : de set_dwn_phi < ..., >;	Grid1D
nde_ff_fun_[.i](inode)	$f_{i+nlo}(\eta, \varphi)$	fun : de set_node_phi < ..., >;	Grid1D
phi_tef(up[.i])	$f_{i+o}(\eta, \varphi)$	fun : de set_up_phi < ..., >;	Grid1D

Les matrices du TEF :

Jacobiennes	sous-matrices	*signification*	TEF-mx
H	H(1 :np, 1 :np)	$\partial_\eta g(\eta,\varphi)$	$A = I - \frac{\delta t}{2}H$
	H(1 :np, np+1 :np+lp)	$\partial_\varpi g(\eta,\varphi,\varpi)$	id
Bb	Bb(1 :np, 1 :mp)	$\partial_\varphi g(\eta,\varphi)$	$B = -\frac{\delta t}{2}Bb$
Ct	Ct(1 :mp, 1 :np)	$\partial_\eta f(\eta,\varphi)$	Ct
	Ct(1 :mp, np+1 :np+lp)	$\partial_\varpi f(\eta,\varphi,\varpi)$	Ct
	Ct(mp+1 :mp+mobs, 1 :mp)	$c_2 = \partial_\eta f_\mu(\eta,\varphi)$	Ct
	Ct(mp+1 :mp+mobs, np+1 :np+lp)	$c_1 = \partial_\varpi f_\mu(\eta,\varphi,\varpi)$	Ct
D	D(1 :mp, 1 :mp)	$\partial_\varphi f(\eta,\varphi)$	D
	D(mp+1 :mp+mobs, 1 :mp)	$\partial_\varphi f_\mu(\eta,\varphi,\varpi)$	D

3 Le découpage du code

Les séquences principales programmées par l'utilisateur sont au nombre de trois :
- la séquence ZINIT est la principale, où le modèle mathématique est décrit, les variables et paramètres initialisés en pseudo-Fortran et les options de calcul choisies ;
- ZSTEER est exécutée en fin de boucle temporelle pour offrir des possibilités de gestion (printout, données, calculs diagnostics particuliers, ... ;
- DIMETAPHI est la séquence de déclaration des paramètres de dimensionnement du modèle ;

De plus, la programmation utilise de nombreuses séquences pour appliquer le principe de non-duplication ; la liste en est donnée ci-après. Cette option de découpage du code de calcul en sous-parties de programme permet à l'utilisateur d'en importer dans sa source de code propre de façon à en modifier certaines sans pour autant se détacher de l'évolution du reste du code.

Enfin, ce découpage du code est optionnel pour certaines parties de calculs spécifiques. Ainsi, le code relatif au filtre de Kalman est activé par un flag logique (**Kalman**, et n'est généré en Fortran que si ce flag est activé (par *sel* ou *selmod*).

3.1 Tableau des séquences

Par ordre de première apparition dans le programme principal, liste des séquences indépendantes des flags logiques.

nom	*contenu*	*dans quels codes*
FTN_opt	implicit simple ou double précision	général
FTN_OPT_Principal	restriction FTN option	principal
LO_UNITS	unité logiques FTN	partout
Definition_Dimension	macros **real**→ **double**	principal, subroutines
DimEtaPhi	paramètres de dimensionnement du modèle	principal, subroutines et autres programmes
Definition_Variable	déclaration des variables et dimensionnements induits	principal, subroutines
Derive_Mac	macros MTN dérivation formelle	principal
Set_Eta_Phi	macros MTN descro modèle	principal
Print_Hlp	construction de Modele.hlp et helps printout	principal
Z_Pr	macro facilité de printout	partout
ZINIT	programmation du modèle	principal
ZCmd_law	loi de commande éventuelle	trajectoire, adjoint
matrice_D	calcul D-Loop	*ibidem*
D_mx	dérivation matrice D	trajectoire, adjoint
matrice_ABCt	dérivation des Jacobiennes H, Bb et Ct	trajectoire, sensibilité et adjoint
A_Mx	dérivation de Jacobienne H	*ibidem*
B_Mx	dérivation de Jacobienne Bb	*ibidem*
Ct_Mx	dérivation de Jacobienne Ct	*ibidem*
Gamma_vec	calcul vecteur Γ	traj., sensib. et adjoint
Omega_vec	calcul vecteur Ω	*ibidem*
OmeGam_sup	analogues *ibidem*	extensions
Borel_vecs	vecteurs fonction de δt	Borel sweep
ZSTEP	(vide) avant incrémentations fin pas de temps	principal
ZSTEER	gestion fin pas de temps	principal
prt_mat	print Jacobiennes et vecteurs	générale
Cout_Psi	Fonction de coût terminal	direct et adjoint
Cout_l	intégrande fonction de coût	trajectoire et adjoint
trans_ABCD	transposition des matrices du TEF	adjoint

Il est toujours possible d'importer une séquence dans son **patch zinproc** pour en modifier le contenu, qui est alors prioritaire dans la génération du Fortan. Ainsi de la séquence (102) ZSTEP, permettant d'introduire un critère sur **dt**, sans incrémenter et en renvoyant à **:ReDoStep :**;

3.2 Séquences supplémentaires

Cette liste complémentaire concerne des parties du code générées sur activation de flags logiques :

nom	*contenu*	*options*
varcov_update	mise à jour matrice de variance-covariance	Obs
dPi_HBbF	sensibilité Jacobienne à un paramètre	dPi_Aspha
dPi_CtD	*idem*	*idem*
dffk_HBb	*idem*	*idem*
SELF-code,IF=Kalman	avance matrice P et calcul de K optimal	Kalman

4 Les algorithmes

4.1 Calcul de la trajectoire

Enchaînement des calculs :

- le ZINIT : description du modèle et initialisations ;
- **début de la boucle temporelle** $\longleftarrow \bullet$;
- La **D-LOOP** : résolution du système implicite $\varphi = f(\eta, \varphi)$;
- calcul des matrices Jacobiennes H, Bb, Ct et D par dérivation symbolique ;
- extensions : calcul des Ψ (Phi_t et Sens_p) ;
- initialisation éventuelle particulière de Φ ;
- sortie des variables (qui sont en phase après la D-loop) ;
- calculs diagnostics (matrice d'avance d'état A^{spha}, fonction de coût J) et sauvegardes pour l'adjoint ;
- test de linéarité de l'avance de φ ($f(t+\delta t) \leftrightarrow f(t) + d\varphi$) ;
- (boucle sur δt si balayage de Borel$\rightarrow$ gain) ;
- calcul des vecteurs et matrices du TEF (intervention de δt) ;
- préparation analogue pour le calcul des extensions ;
- **appel à ker pour la résolution** ;
- séquence ZSTEP (pour gestion de δt ;
- écriture fichier de $\delta t, \delta\eta$;
- avance η et $t, \delta t$ (et extensions) ;
- passage par la séquence ZSTEER ;
- **fin du pas de temps, boucle** $\bullet \nearrow$;
- fermeture fichier et arrêt.

Le ZINIT

Cette séquence où l'utilisateur définit les modèles, fournit les conditions initiales, et explicite les options de calcul ne présente de difficulté qu'en ce qui concerne le langage, c'est-à-dire le traitement par **Mortran** du code entré. Les explications en sont fournies dans un autre cahier : "Les macros Mortran"[1]. Nous ne donnerons ici que les principes.

On considère le code suivant, qui déclare les modèles de cellule de l'exo Lorenz :

```
!%%%%%%%%%%%%%%%%%%%%%%%%
! Cell definition
!%%%%%%%%%%%%%%%%%%%%%%%%

set_eta
< var: eta_courant_L,
  fun: deta_conv = pi_prandtl*ff_T_czcx
                              - pi_prandtl*eta_courant_L;
  var: eta_T_czcx,
  fun: deta_energy_zx = - ff_Psi_Tsz + pi_Rayleigh_ratio*ff_courant_L
                                              - eta_T_czcx;
  var: eta_T_sz,
  fun: deta_energy_z = ff_Psi_Tczcx - pi_wave_nb_ratio*eta_T_sz;

>;
```

On a ainsi défini trois variables d'état, chacune d'elles associée à un "modèle", c'est-à-dire un équation d'évolution $\partial_t \eta = g(\eta, \varphi)$. L'expression $g()$ sera dérivée par rapport aux

[1]en progression.

composantes de η (matrice H) et φ (matrice B_b). Le calcul de ces dérivées est écrit en utilisant les symboles standards de variables d'état et sa formule correspondante :

```
!---------------------------------------------------------
! matrice    H(i,j) = dEta(i)/dEta(j)
!---------------------------------------------------------
       do j=1,np
       <
         ;mult[i]=1,np :  H([i],j) = f_d(deta_tef[i])(/eta(j));
       >;
```

En premier lieu, le symbole **np** de la **mult** est remplacée par "3", grâce à des macros travaillant sur le code de **dimetaphi** (explicite ou généré dans la 102). Ensuite, la macro **mult** génère trois lignes avec incrémentation de [i] de 1 à 3 (La **set_eta** a mis un compteur à la valeur 1).

Une des trois lignes est à présent

```
  H((2),j) = f_d(deta_tef(2))(/eta(j));
```

les macros **var :** et **fun :** on l'effet

- d'associer **deta_tef(2))** à la formule déclarée en deuxième position' :

```
H((2),j) = f_d( - ff_Psi_Tsz
                     + pi_Rayleigh_ratio*ff_courant_L
                                         - eta_T_czcx)
                                                 (/eta(j));
```

- ensuite, chaque symbole déclaré **var** dans le modèle sera remplacé par la composante **eta(.)** correspondante – et **ff(.)** pour les **set_phi** ;

```
H((2),j) = f_d( -ff(2) +pi_Rayleigh_ratio*ff(3) -eta(2))(/eta(j));
```

- il reste alors a effectuer la dérivation **(/eta(j))**.

Dans la version historique du **ZINIT**, on déclarait l'équivalence entre un nom de formule (Phi_TEF) et son expression avec un set_f. Avec le langage plus avancé, on a simplelement une étape supplémentaire faisant intervenir un troisième symbole déclaré par **fun : symbole = expression** ;.

Tout le jeu des macros générées par les **set, var et fun** s'explique par cet objectif de remplacement, la complexité apparente des macros tenant aux règles de parcours des symboles dans Mortan. Bien sûr, la séquence décrite n'a pas lieu dans cet ordre, et chaque ligne [i] générée est d'abord travaillée par des macros remplaçant [i] par un compteur, puis soumise aux macros générées par le **ZINIT**, avant le passage au compteur suivant, avec procédure d'arrêt, etc, ce qui fait qu'un **&T3** de Mortan sur un calcul est un peu pénible à suivre – ce n'est pas une raison pour s'en priver.

De plus, les macros génèrent du code informatif (**help code**) et appellent une routine **prohlp** qui a pour travail de conserver ce lien entre composante de vecteur et symbole-utilisateur pour informer les sorties du code (et repérer les composantes des vecteurs et matrices imprimées). Enfin, un **modele.hlp** rassemble les formules du modèle.

On conçoit alors d'emblée l'importance de la définition d'un langage, puisque non seulement il permet de séparer la formulation algorithmique du code des symboles du calcul, mais de plus il permet de générer du code Fortran supplémentaire pour vérifier, conserver *et transmettre de l'information tout au long de la chaîne de calcul, dans les print_out et dans les fichiers résultats. Au moment où ces lignes sont écrites, seul Mortran offre d'aussi vertigineuses potentialités*[1].

La D-Loop

La résolution de l'équation implicite des transferts $\varphi = f(\eta, \varphi)$ doit être effectuée dès que l'état est connu, avant calcul des Jacobiennes – le modèle non linéaire faisant *a priori* dépendre ces matrices de φ et de η.

C'est le cas dès la sortie de **ZINIT** – les C.I. étant données – et après le calcul d'avance d'état (après résolution par **call ker** et $\eta = \eta + \delta\eta$. Il n'est donc pas étonnant que la boucle temporelle débute par cette séquence. Par ailleurs, le calcul itératif de cette équation implicite doit éviter de possibles singularités en φ – intervention de $\frac{1}{ff([i])}$ par exemple, ce qui pourra exiger de fixer des valeurs initiales particulières à certaines composantes de transfert, d'où l'algorithme suivant.

En initialisation de la variable φ (tableau **ff**) lors de l'entrée dans la boucle d'avance de la trajectoire, on conserve les valeurs données par ZINIT.Par ailleurs, lors de la résolution du système implicite $\varphi = f(\eta, \varphi)$, il est numériquement nécessaire que la solution reste indépendante de l'ordre de calcul des composantes du vecteur φ. Ceci se fait par recopie de **ff** dans le tableau **phiz** et calcul de **ff(.) = f(eta,phiz)**. La fonction $f()$ est en correspondance symbolique avec **Phi_tef[.]**[2].

La relaxation est ensuite effectuée par une méthode de Newton-Raphson, η étant fixé :

$$\begin{aligned} \varphi_0 + \delta\varphi &= f(\eta, \varphi_0) + D\delta\varphi \\ \to \delta\varphi &= [I - D]^{-1}[f(\eta, \varphi_0) - \varphi_0] \end{aligned}$$

avec $D = \partial_\varphi f(\eta, \varphi)|_{\varphi_0}$. La variable vectorielle φ_0 est dans le tableau **phiz** (routine **newt**). Il faut donc calculer la matrice D en dérivant **f**(φ_z) en fonction de **phiz**, d'où les macros MTN transformant les formulations de **Phi_tef(ff)** en **Phi_tef(phiz)**, qui sont annulées en sortie de la D-Loop ("retour à ff").

La relaxation est itérée jusqu'à satisfaction du critère $f(\varphi) - \varphi_0 < TOLF$ ou $\delta\varphi < TOLX$, ou abandon après **NTRIAL** essais (avec diagnostic). En option **Debug**, on a en plus une estimation du conditionnement de $[I - D]$ et sortie de chaque itération.

Fin du pas de temps et sorties des variables

La résolution du système implicite en φ termine le calcul d'un pas de temps, puisqu'alors les variables η et φ sont en phase. Les fichiers **.data** sont alors abondés d'une ligne pour chaque vecteur (avec la variable **time** en tête :

- **ff** vecteur φ et **obs** vecteur μ ;
- **eta** vecteur η, suivi du logarithme du Wronskien (cf SLT_Al1 28) ;
- **Phi_t** et le couple **Psi_t, PsiMu_t** des sensibilités aux variables ;
- **sensibilités paramétriques** dans les tableaux **Sens_p** et **Sigma_p** ;
- **dnEta**, la variation de l'état, correspond en sortie (avec la valeur de l'incrément **dt**) à l'avance entre **time** et **time+dt** ;
- **autres incréments** de même que ci-dessus, **dPhi_t** et **dSens_par** sont sortis ;
- de manière provisoire pour vérifications, on sort aussi **Phi_t+dPhi_t** ;

Tous les résultas sont écris dans des fichiers **.data** assortis éventuellement d'un suffixe différent (**osuffix**) comme décrit dans le Manuel pratique.

[1] il est utile ici de citer un cadre du CNES qualifiant Mortan :
"c'est chiant, c'est vieux,..., on a pas trouvé mieux" (dixit vip, Juillet 2006).

[2] cf dérivation formelle 4.1.

La dérivation des Jacobiennes

Un "modèle", associé à une variable, correspond à chaque ligne d'une Jacobienne. La formule associée devra être dérivée par rapport à toutes les variables $\eta, \varphi, \mu, \varpi$ dans des matrices différentes. On distingue donc dans le code la génération d'une formule [i] en correspondance avec le modèle d'une variable [i], et sa dérivation (\var(j)) qui est effectuée dans une boucle sur (j), comme explicité maintenant.

Le calcul des coefficients des matrices Jacobiennes est effectué dans une boucle Fortran par lignes, chaque ligne correspondant á un "modèle" différent :

```
!---------------------------------
! Matrice d(i,j) = dPhi(i)/dPhi(j)
!---------------------------------
        do j=1,mp
        < mult [i]=1,mp : d([i],j) = f_d(Phi_tef[i])(/ff(j));
        >;
;
```

Une ligne est calculée par l'expansion du code **mult** qui génère $i = 1, mp$ instructions Fortran. Le symbole **Phi_tef[i]** est ainsi remplacé d'abord par l'instanciation **Phi_tef[1]**, ce dernier symbole lui-même remplacé par sa définition dans ZINIT, c'est-à-dire soit l'expression à droite du signe égal dans le cas d'une déclaration explicite de type **Phi_tef[1] = expression FTN**, soit par un **set_phi** qui lui-même aura généré une déclaration explicite de remplacement.

Enfin, la macro **f_d(S1)(/S2)** effectue la dérivation de l'expression mathématique dans **S1** par le symbole **S2** - soit ici **ff(j)**. Remarquons à ce sujet que la dérivation d'une expression fonction de ff([i]) par une variable indicée – ici ff(j) – est effectuée en dérivant par la chaîne de caractères ff(), et le résultat multiplié par la fonction implicite **F_delta([i],j)** nulle si $[i] \neq j$, égale à 1 sinon. Autrement dit, lorsqu'on dérive une expression par ff(j) dans une boucle **Do** portant sur j – c'est le cas de l'exemple ci-dessus, le code Fortran est identique, seule la valeur de **F_delta** annule ou non cette dérivée.

cas du Grid 1D

On a d'une part les transferts **dwn** et **up** qui sont semblables aux standards :

```
!---------------------------------
! Matrice d(i,j) = dPhi(i)/dPhi(j)
!---------------------------------
;
        if m_dwn .gt.0
        < do j=1,mp
          <  ;mult [i]=1,m_dwn :     d([.i],j) = F_D(Phi_tef[i])(/ff(j));
          >;
        >;
        do j=1,mp
        < do inode=1,m_node
          < ;mult [i]=1,m_mult :
              d(k2j_[.i](inode),j) = f_d(nde_ff_fun_[.i](inode))(/ff(j));
          >;
        >;
        if m_up .gt.0
        < do j=1,mp
          <  ;mult [i]=1,m_up :
                      d(joffsetup+[.i],j) = F_D(Phi_tef(up[.i]))(/ff(j));
```

```
        >;
      >;
;
```

Pour les **nodes**, on a **m_node** séquences d'expressions identiques ne différent que de la valeur de l'indice de boucle **inode**. C'est encore la fonction **F_delta(inode,j)** qui annulera ou pas la dérivée.

les autres Jacobiennes et le passage aux matrices et vecteurs d'avance

Les matrices Jacobiennes A,Bb,Ct sont calculées de manière semblable à D. Le calcul du vecteur **ff** aussi, mais sans dérivation.

Dans la mesure où ces calculs de Jacobiennes interviennent dans l'adjoint, leur code est regroupé pour chacune dans une séquence (cf tableau 3.1).

Enfin pour finir avec les matrices du TEF, les matrices $A = I - \frac{\delta t}{2}H$ et $B = -\frac{\delta t}{2}Bb$ sont effectuées dans le corps de **principal** :

```
!=====================================================
! Complément de calculs dépendants de dt
!=====================================================
! build a mx(init a I) from h : a = I -dt/2*h
      call scamat(-dt/2.,h,hdt2m,n,np,n+lp,np+lp);
      call diagmat(one,a,n,np+lp);
      call matadd(a,hdt2m,a,n,np,n+lp,np+lp);
;
! Build TEF B mx from Jacobian Bd
      call scamat(-dt/2.,Bb,B,n,n,m,m);
```

Deux autres séquences concernent enfin le calcul des vecteurs Γ et Ω :

```
!*KEEP,Gamma_vec.
!---------------------------------------------------
! vecteur gamma avec : (dt)eta = deta_tef(eta,ff,t)
!---------------------------------------------------
      mult[i]=1,np : gamma[i] = dt*(deta_tef[i]
                                  + 0.5*dt*(f_d(deta_tef[i])(/time))
                                 );

!*KEND.
!*KEEP,Omega_vec.
!----------------------------------------------
! vecteur omega avec : Phi = Phi_tef(eta,ff,t)
!----------------------------------------------
     ;mult[j]=1,mp : omega[j] = dt*f_d(Phi_tef[j])(/time);

!*KEND.
```

où seule une dérivation par rapport à la variable **time** est effectuée.

Ceci termine le calcul de l'ensemble des vecteurs (2) et matrices (4) nécessaires au calcul de la trajectoire (de t à $t + \delta t$).

La matrice A^{spha}

La matrice d'avance du système d'état – hors TEF, c'est-à-dire lorsque les transferts ont été éliminés – présente un intérêt théorique, car dans ce cas on se retrouve dans un cas largement étudié de l'analyse des systèmes dynamiques. Cette matrice permet par exemple le calcul de gains de rétroaction non-stationnaires (cf Effluents). Elle est également nécessaire pour faire avancer le calcul de la matrice de var-covar de Kalman.

Lorsque la matrice $(I-D)^{-1}$ est régulière, la matrice dîte d'avance d'état est $A^\sharp = H - B_b(I-D)^{-1}C^\dagger$ (tableau FTN Aspha) :

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
! Calcul de la matrice d'avance dans l'espace de phase
!       d_t Eta = [ A + B.(I-D)-1.Ct ] Eta
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
       call omatmlt(m,m,n,un_moins_d,mp,ct,mp+mobs,ctbuf,mp);
       call omatmlt(n,m,n,Bb,np,ctbuf,mp,aspha,np);
       call omatadd(n,n,h,np,aspha,np,aspha,np);
       if (Zout) write(loaspha,1001) time,((aspha(i,j),i=1,n),j=1,n);
! Calcul du Wronskien [d_t Log W = Tr(Aspha)]
       oTrace_adv = 0.; <i=1,n; [oTrace_adv :+aspha(i,i)]; >;
       oInt_Tradv = oInt_Tradv + dt*oTrace_adv;
! -----------------------------------------------------------------
```

On voit que l'on en profite pour calculer le Wronskien, c'est-à-dire le logarithme du déterminant du propagateur, qui avance par la trace de $A^\sharp$ (cf SLT_All 28).

Borel sweep et Boreleig-Boreigrun

On explique ensemble ces deux algorithmes parce qu'ils résolvent chacun, bien qu'avec des méthodes orthogonales, le problème de la détermination du gain, réponse et effet de rétroaction stationnaire (cf Effluents).

Le principe par **Borel sweep** est de déterminer numériquement un gain en profitant de la formulation identique de l'avance par $\frac{\delta t}{2}$ et la transformation de Borel du système local (considéré alors comme autonome) avec la variable $\tau \leftrightarrow \frac{\delta t}{2}$.

On va alors balayer en δtune réoslution par **ker** et conserver le gain $g_B(\tau)$ fourni par **ker**. Pour limiter les effets de ce balayage dans **principal** . on renvoie à une routine :

```
! --  Balayge de Borel en dTau
       call Bsweep(H,Bb,Ct,d,deta_B,deta_tB,dPhi_tB,dt,index_ff_gain,
            itau_max,tau_B_ini,tau_B_mult,time,istep,.false.);
```

à qui on transmet les Jacobiennes. Il faut cependant aussi tenir compte des vecteurs dépendants de δt, dont on calcule et transmet la dérivée temporelle :

```
!*KEEP,borel_vecs.
!          vector computations for borel sweep
!------------------------------------------------------------
! sequence de calcul des parties independantes de dt
! des vecteurs gamma omega pour exportation B-sweep
!------------------------------------------------------------

!----------------------------------------------------------
! pour vecteur gamma avec : (dt)eta = deta_tef(eta,ff,t)
!---------------------------------------------------------
       mult[i]=1,np : deta_B[i] = (deta_tef[i]);
       mult[i]=1,np : deta_tB[i] =f_d(deta_tef[i])(/time);
      ;mult[j]=1,mp : dPhi_tB[j] = f_d(Phi_tef[j])(/time);
```

```
!*KEND.
```

Le principe dans **ker** de la détermination du gain consiste à éliminer toues les composantes de transferts sauf une (spécifiée par le pointeur **index_ff_gain**). La variable **gain** donne en sortie la valeur de $\tilde{g}_B(\tau)$ - cf **ker**. On a donc en sortie $\mathcal{B}[g_B(\tau)$, que l'on pourra inverser par fit etc selon la fameuse horreur borelienne.

La routine **Boreleig** (pour Borel par eigen-elements) calcule en une seule fois les éléments propres des matrices $A^\sharp$ et $A^\flat$ (cf Effluents (114)) pour permettre le calcul en off-line des éléments de la fonction du temps physique (constituée de exp(), sin() et cos()). **Boreigrun** utilise les coefficients calulés par la Boreleig pour en faire une série num érique en balayant la variable temporelle pour tracer des graphes standards (mais rien n'empèche d'utiliser la fonction temporelle lorsqu'elle n'est pas trop longue (Gnuplot est limité à quelques dizaines de caractères). Boregrun se trouve dans le patch **gsrun**).

Enfin, on peut utiliser avantageusement **PAW** pour visualiser les fonctions statiques telles qu'elles sont calulées le long d'une portion de trajectoire en utilisant la routine (hors patch) **gains.f** (voir directement le code FTN). On obtient une **surf2z**, voire un exemple d'utilisation dans le cahier "Lorenz et gains".

4.2 La routine oKer, solveur du système

Ayant en argument d'entrée les quatre matrices d'avance $A, B, C^\dagger, D$ du TEF et les vecteurs connus Γ, Ω, le sous-programme **oker** effectue la résolution du système (4) en deux temps : élimination des cellules menant au système dit des "couplages" entre les transferts, résolution des transferts, et enfin "descente" pour résolution des états. On a en sortie les variations $\delta\eta$ et $d\phi$.

En réalité, **oker** étend cette résolution à toutes les colonnes supplémentaires aux deux vecteurs du TEF, soit **nwp** colonnes supplémentaires : (Γ_x, Ω_x), exploitant ainsi l'efficacité de la méthode de résolution de système linéaire de LaPack (routine SGESV).

Enfin, l'option de balayage par **Borel-sweep** utilise une colonne de plus, parce que le gain lié à une composante i de transfert (integer **i_gain**) s'obtient par résolution du système de couplage pour le vecteur connu $\Gamma_g =| 1_i >= [0, 0, ..., 1, 0, ...0]^\dagger$ avec 1 en position i. Le gain se trouve en position i du vecteur solution.

Les autres arguments d'entrée (**ZPrint, eta, etax**) ne servent qu'à l'impression, et on a en sortie les deux arguments d'information LaPack (**infoa, infores**).

La seule difficulté de **oker** consiste ainsi au rangement des colonnes des vecteurs connus et cette gestion du calcul optionnel du gain. Les matrices en entrée sont elles mêmes possiblement des sous-matrices; on esquive cette difficulté en recopiant en entrée les blocs-matrices utilisés dans des tableaux intermédiaires.

Les calculs et tableaux sont organisés comme suit :

- "vecteurs" connus rassemblés dans la matrice **bggx** dans l'ordre **bw, gamma, gammx**; de cette manière, la "montée" déterminant le système des couplages fournit à la fois la matrice $A^{-1}B$ et les variations découplées $A^{-1}\Gamma, \Gamma_x$ après l'appel à un premier **sgesv** (avec le flag LaPack **infoa**). On a ainsi **np+1+nwp** colonnes de résultat;
- il reste à multiplier par $C^\dagger$ pour obtenir la matrice de couplage d'une part, les évolutions insensibles $(C^\dagger A^{-1}\Gamma, \Gamma_x)$ d'autre part

```
!  ** calcul de Ct*[A-1*B&gamma&gamx] = Ct*BGGx *****

call omatmlt(mp,np,mp+1+nwp,ct,mp+mobs,bggx,np,ctbggx,mp);
```

```
! *********** calcul de matrice de couplage **
!    couplage = I - (-Ct) A-1B - D

CALL omatadd(mp,mp,c_unit,mp,ctbggx,mp,unplusctamb,mp);
CALL omatsub(mp,mp,unplusctamb,mp,d,ldd,couplage,mp);
!                                       ========

if Zprint
<     TITLE='>ker : Matrice de couplage';
      CALL printmat(title,couplage,mp,mp,mp,mp,2,2);
>;

! *********** calcul de dphi_ins  ***********
!    dphi_ins = Omega - (-Ct)A-1.Gamma

 CALL omatadd(mp,1+nwp,ctbggx(1,mp+1),mp,omegxx,mp,dphixx,mp);

if Zprint
<     TITLE='>ker : dphi_ins';
      CALL printmat(title,dphixx,mp,mp,1+nwp,1+nwp,0,2);
>;
```

– on est à présent “en haut de navette” dans le système de couplage, avec un “vecteur connu” **dphixx** de **1+nwp** colonnes. Ici intervient l'option Borel-sweep, avec création du vecteur $\mid 1_i >$ rangé en **1+nwp+1**

```
if (i_gain.EQ.0)
< call sgesv(mp,nwp+1,couplage,mp,iPiv,dphixx,mp,infores);
  if infores.ne.0
  < print*,'*** SGESV(couplage,dphixx) error: ',infores;
  >;
>else
< call veczero(dphixx(1,nwp+2),mp);
  dphixx(i_gain,nwp+2)=one;
  call sgesv(mp,nwp+2,couplage,mp,iPiv,dphixx,mp,infores);
  if infores.ne.0
  < print*,'*** SGESV(couplage,dphixx+gain) error: ',infores;
  >;
  gain = dphixx(i_gain,nwp+2);  "en realite, =1/(1-g)"
>;
;
```

– descente de navette avec calcul de $A^{-1}B\,[\delta\varphi, \delta\varphi_x]$

```
! *****************************************************
! *********** calcul de a-1b*dphi,x  ***********
  CALL omatmlt(np,mp,1+nwp,bggx,np,dphixx,mp,ambdphixx,np);
! *****************************************************
```

– et enfin calcul des avances $\delta\eta, \delta\eta_x$ du genre

```
! Calcul avance etat
DO I=1,np
< dneta(i)=agamdt(i)-ambdphixx(i,1);
  if (ZPRINT)
  write(*,9001)i,agamdt(i),ambdphixx(i,1),dneta(i),eta(i),
                                    EtaNam(i)(1:lenocc(EtaNam(i)));
>;
```

```
        9001;format(2x,I3,4(1Pe12.4),1x,A);
```

et des extensions, avec recopie dans les arguments de sortie;

La routine n'est pas parfaitement optimisée, elle fait un compromis entre efficacité et clarté-robustesse aux développements en conservant quelques copies inutiles, mais permettant des impressions de debugging.

En sortie de oKer

On range le calcul de l'avance $\delta\eta$ dans son **dres.data**, et on avance Etat

```
! Calcul avance etat,x
! --------------------
call omatsub(np,1+nwp,bggx(1,mp+1),np,ambdphixx,np,dnetax,ldeta);

if Zprint
< print9000;print*,' ---- avance d Etat - State advance -----';
  print *,' i,  A-1gamdt(i), A-1Bdphi(i),   deta(i),     eta(i) Var name';
 DO I=1,np
  < write(*,9001)i,bggx(i,mp+1),ambdphixx(i,1),dnetax(i,1),etax(i,1),
                                        EtaNam(i)(1:lenocc(EtaNam(i)));
  >;
>;
```

de plus, on utilise le vecteur $\delta\varphi$ pour faire avancer les transferts à seule fin de contrôle des calculs de la trajectoire :

```
!-----------------------------------------------------------
! pour test linearite : ffl pour next step
     do i=1,m
     < ffl(i) = ff(i) + dphi(i);
     >;
```

Ceci pour comparer l'avance linéaire des transferts aux transferts calculés par l'algorithme du TEF qui sont $\varphi = f(\eta, \varphi)$. On donne ici le test effectué en fin de pas de temps :

```
! ==========================================================
! test linearite ffl - ff
! ==========================================================
if (istep.gt.1)
< res=0.; <io=1,m; res = res +(ffl(io)-ff(io))**2/max(one,ff(io)*ff(io)); >;
  if (res .gt. TOL_FFL)
  < print*,'*** linearity pb: res > TOL_FFL at istep',istep,res,' > ',TOL_FFL;
    write(*,*)'  i,   ff(i)   , ff(i)-ffL(i), var Name';
    do io=1,m < write(*,'(i4,2(1pe12.5,1x),a)')
                                   io,ff(io),ff(io)-ffl(io),PhiNam(io); >;
    if (idxerror.eq.0.) idxerror = -1;
  >;
>;
```

où on voit qu'il repose sur l'écart quadratique "quasi-relatif" entre les deux calculs de φ, comparé au seuil **TOL_FFL** fixé soit dans ZINIT, soit à son défaut (10^{-6}), ceci remarquons le indépendamment de la précision (simple ou double) des calculs[1].

[1] ce qui est la méthode correcte pour confronter les deux types de calculs.

4.3 Calcul des sensibilités des variables

Comme on l'a expliqué dans le formulaire liminaire, l'ensemble de ces calculs de sensibilité des variables η, φ, μ sont regroupés colonne par colonne dans des tableaux en extension des vecteurs de variables ;

Les extensions

Description des équivalence et dimensions ;
On va séparer l'explication des calculs des sensibilités aux variables et aux paramètres.

Calcul des sensibilités à des variables

On a d'abord le calcul du Γ_Φ

```
! ----------------------------------------------------------
! extension Gamma_Phi_t = dt(H Phit + Bb Psit [+source])
     if nxp.gt.0
     < call omatmlt(n,n,nxp,H,np,Phi_t(1,1),np,gam_Phit(1,1),np);
       call omatmlt(n,m,nxp,Bb,np,Psi_t(1,1),mp+mobs,bobuf,np);
       call omatadd(n,nxp,gam_Phit(1,1),np,bobuf,np,gam_Phit(1,1),np);
       call oscamat(dt,n,nxp,gam_Phit(1,1),np,gam_Phit(1,1),np);
     >;
```

Ensuite, le traitement du cas 4 (g/(1-g)) est effectué avant résolution et avance :

```
     if nxp.gt.0
     < jo=0;
       Do ko=1,nxp
       < if (istep.ge.ko_p_ini(ko).and.ko_p_typ(ko).eq.4) jo=1;
       >;
! %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!  Source pour propag. de transfert marche : gam_x  -> + |BbDm1>_k dt
!             de type g/1-g
! -------------------------------------------------------------------
       if jo.eq.1
       < call matmlt(Bb,un_moins_D,bobuf,n,np,m,mp,nxp,nxp);
         call scamat(dt,bobuf,bobuf,np,np,nxp,nxp);
         Do ko=1,nxp
         < if ko_P_typ(ko) .eq. 4
           < call oscamat(step_psi(ko),n,1,bobuf(1,ko_p_kvar(ko)),np
                                    ,bobuf(1,ko_p_kvar(ko)),np;
             call matadd(bobuf(1,ko_p_kvar(ko)),gam_Phit(1,ko)
                                    ,gam_Phit(1,ko),n,np,1,nxp);
           >;
         >;
       >;
     >;"End nxp"
! =================================
!      call oker
! =================================
! avance Phi_t
! ------------
   if (nxp.gt.0) call matadd(Phi_t,dPhi_t,Phi_t,np,np,nxp,nxp);
```

Remarquer la multiplication par step_psi de l'amplitude bobuf pour obtenir la réponse effective à un step d'amplitude différengt de l'unité. Il reste à compléter la partie Ψ, avec les sources pulsées :

```
! fin du pas de temps :
! ================================
!      D-Loop, Jacobiennes, puis :
! ================================
! Extension : calcul de (I-D) Psi_t = Ct Phi_t  [+source]
! ---------------------------------------------------------
     if nxp.gt.0
     <
! --- suivant indications des ko_p var, raz Phi_t, et sources sensib
        call pSteer("in:" np,nxp, Bb,np, un_moins_d,mp,
                          ZPrint,lp,mobs,EtaNam,PhiNam,ParNam,
                                                 "out:" Phi_t,np);
! --- calcul de Ct Phi_t [+source]
       call omatmlt(m,n,nxp,Ct,mp+mobs,Phi_t(1,1),np,Psi_t(1,1),mp+mobs);
! ---
!    -- add perpetual unit source for type 3 pert
       Do ko=1,nxp
       < if ko_P_typ(ko).eq.3 .and. istep.ge.ko_p_ini(ko)
         < call oscamat(step_psi(ko),m,1,s_unitm(1,ko_p_kvar(ko)),mp
                                                         ,T_tmp1(1,1),mp);
           call omatadd(m,1,T_tmp1(1,1),mp
                                  ,Psi_t(1,ko),mp+mobs,Psi_t(1,ko),mp+mobs);
         >;
       >;
! --- Calcul Dm1Psit
    call omatmlt(m,m,nxp,un_moins_d,mp,Psi_t(1,1),mp+mobs,Psi_t(1,1),mp+mobs);
;
     >;"End nxp"
```

après quoi on complète par une multiplication par la $C^{\dagger}$ avant de finir avec $(I-D)^{-1}$. Le buffer T_tmp1 ne sert qu'à prendre une amplitude de step définie dans ZINIT au lieu de l'unité. Autre perturbation, autre source, l'ensemble étant géré dans la routine **PSteer**, avec par exemple :

```
!     --- source pour perturbation-marche
       if io.eq.1
       < print*,' --- (re) setting step(s) on transfer at:',time,istep,' -----';
         print*,'  -- for 1/(1-g) response-type ------------------------';
         print*,'  -  (setting Unit Source: Dm1 Psi_t = CtPhit + I)   ---';
         Do ko=1,nxp
         <
!    -- RAZ Phi_t, Psi_t si nouveau depart Tr-Heaviside
           if ko_P_typ(ko).eq.3
           < if Zko_p(ko)  " RAZ Phi_t, Psi_t "
             < call veczero(Phi_t(1,ko),np); call veczero(Psi_t(1,ko),mp);
              if ZPrint
              < print*,'   for Tr #:',ko_p_kvar(ko),' on:',ko,' Psit colon ---';
              >;
             >;
```

```
        if ko_P_typ(ko).eq.3 .and. istep.ge.ko_p_ini(ko)
        < call matadd(s_unitm(1,ko_p_kvar(ko))
                              ,Psi_t(1,ko),Psi_t(1,ko),m,mp,1,nxp);
        >;
      >;"end 3"
    >;
  >;
```

et même final que pour la source précédente.

On ne décrit pas ici la partie remise à zéro et initialisation, qui sont effectuées en fonction du time-flag suivant :

```
  Do ko=1,nxp
  < Zko_p(ko)=(istep.eq.ko_p_ini(ko)
                .or. mod(istep-ko_p_ini(ko)+1,ko_p_mod(ko)).eq.1);
  >;
```

qui est **.True.** pour les **istep = ini** [**mod**], correspondant aux **at every**, qui vaut en particulier **.False.** pour la valeur de -1 donnée à **ko_p_mod** (c'est-à-dire en l'absence de **every**. Il faut cependant traiter à part le cas du premier pas de boucle (c'est-à-dire absence de **at** et donc de **every** qui doit conduire à une action au premier pas.

4.3.1 Calcul des sensibilités aux paramètres

On a deux calculs de "matrice connue" à effectuer, simplifiés par le fait que la sous-matrice $A_\varpi = I$.

La fin du pas de temps résoud l'équation (cf 10)

$$(I - D)\Psi_\varpi = C^\dagger \Phi_\varpi + C^\dagger_\varpi \tag{12}$$

de la façon suivante :

```
! *** Partie parametres : calcul de (I-D) Psi_p = Ct Phi_p + Ct_pi
! ----------------------------------------------------------------
     if lp.gt.0
     < Zko_p(nxp+1)=(istep.eq.ko_p_ini(nxp+1)
                  .or. mod(istep-ko_p_ini(nxp+1)+1,ko_p_mod(nxp+1)).eq.1);
       if Zko_p(nxp+1)
       < call veczero(Phi_p(1,1),np*lp);
            print*,' ---  resetting sensitivity to Parameters at:',time,istep;
       >;
       call omatmlt(m,n,lp,Ct,mp+mobs,Phi_p(1,1),np,Psi_p(1,1),mp+mobs);
       call omatadd(m,lp,Ct(1,np+1),mp+mobs,Psi_p(1,1),mp+mobs
                                                   ,Psi_p(1,1),mp+mobs);
!  -- Dm1Psip
       call omatmlt(m,m,lp,un_moins_d,mp,Psi_p(1,1),mp+mobs,Psi_p(1,1),mp+mobs);
! --- partie Obs : Psi_mu =  c1 + c2 Phi_p + d_fmu Psi_p
       if mobs.gt.0
       < call omatmlt(mobs,n,lp,Ct(mp+1,1),mp+mobs,Phi_p(1,1),np
                                                 ,Psi_p(mp+1,1),mp+mobs);
         call omatmlt(mobs,m,lp,D(mp+1,1),mp+mobs,Psi_p(1,1),mp+mobs
                                 ,Ctbuf,mobs);
         call omatadd(mobs,lp,Ctbuf,mobs,Psi_p(mp+1,1),mp+mobs
                                                 ,Psi_p(mp+1,1),mp+mobs);
         call omatadd(mobs,lp,Ct(mp+1,np+1),mp+mobs
```

```
                              ,Psi_p(mp+1,1),mp+mobs,Psi_p(mp+1,1),mp+mobs);
     >;
    >;"End lp"
! ------------------------------------------------------------
! **************   Ecriture resultats sur fichier  ********
```

Où on voit que c'est les **ko_p_ (nxp+1)** qui gèrent les **at . every .** de l'ensemble des paramètres (qui sont donc synchronisés contrairement aux sensibilités aux variables).

La partie calcul de Γ_ϖ se calcule par (cf 10)

$$\Gamma_\varpi = [H\Phi_\varpi + H_\varpi + B_b\Psi_\varpi]dt \tag{13}$$

avec sa programmation :

```
! --- partie parametres ---
! ------------------------
! extension Omega du Psi_p
    if lp.gt.0
    < call veczero(omeg_Psip(1,1),mp*lp);  "Omega vec null"

! partie Gam_p = [H Phi_p + H_pi + Bb Psi_p] dt
! ---------------------------------------------
      call omatmlt(n,n,lp,H,np,Phi_p(1,1),np,gam_Phip(1,1),np);
      call omatmlt(n,m,lp,Bb,np,Psi_p(1,1),mp+mobs,bobuf,np);
      call omatadd(n,lp,gam_Phip(1,1),np,bobuf,np,gam_Phip(1,1),np);
      call omatadd(n,lp,gam_Phip(1,1),np,H(1,np+1),np,gam_Phip(1,1),np);
      call oscamat(dt,n,lp,gam_Phip(1,1),np,gam_Phip(1,1),np);
    >;"End lp"
```

qui fixe aussi à zéro la matrice Ω_ϖ.

La résolution est prise en compte avec les extensions aux matrices A et $C^\dagger$. La routine **oker** voit en effet toutes les extensions :

```
! +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
! ---------------------------------------------------------------------
   CALL oker("in"   n,m,nxp+lp,  A,np,B,np,Ct,mp+mobs,D,mp+mobs,
                                 Gamma,np,Omega,mp,Eta,np,ff,mp,
                            lp,mobs,EtaNam,PhiNam,ParNam,zprint,
                 "out" dnEta,dPhi, "in" 0, "out" gain,infoa,infores);
! ---------------------******------------------------------------------
! +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

et traite, en plus des vecteurs Γ, Ω du TEF, **nxp+lp** colonnes, correspondant aux **nxp** sensibilités aux variables demandées et aux **lp** sensibilités aux paramètres. La liste des symboles est rangées dans le tableau **ParNam(1 :lp)** lors du traitement dans **ZINIT** de la **Free_parameters** liste.

Remarque : Le remplacement de **ker** *par* **oker** *correrspond à l'adoption de la norme LaPack des dimensions de matrices (Mx,LDMx), c'est-à-dire la matrice suivie du nombre des lignes décarées dans* **dimension Mx(LDMx,*)** ;

L'extension au vecteur des observables

Avec la **version 102** apparait un nouveau vecteur μ dît des observables, en supposant qu'il s'agit de transferts n'ayant aucune rétroaction sur les variables du TEF (matrices

B_b nuules). On suppose de plus (au moins provisoirement) que les composantes sont indépendantes ($D_\mu = 0$).

Tout ceci ne modifie donc en rien la résolution, mais on a à calculer en plus une série de **mobs** fonction explicites en plus, à la fin d'un pas de temps :

```
! ---   Calcul des Observables-probes ---------------------
! ------------------------------------------------------
;mult [j]=1,mobs : ff(mp+[.j]) = (Phi_tef(mp+[.j]));
```

on peut ainsi constater que l'on place ce vecteur à la suite de **ff(.)**, bien que les fichiers **tr et obs.data** restent séparés par raison de commodité.

Bien sûr, ces obervables sont sensibles aux variables du TEF et par la même, directement ou non, aux **Free-parameters**. On a donc besoin de sous-matrices Jacobiennes définies dans (10), programmées comme suit :

```
 mult [i]=1,mobs : d(mp+[.i],j) = f_d(Phi_tef(mp+[.i]))(/ff(j));             <<< D_mx

!------------------------------------
! matrices Ct(i,j) = dFi(i) / dEta(j)
!------------------------------------
        do j=1,np
        < mult[i]=1,mp : ct([.i],j) = f_d(Phi_tef[i])(/eta(j));
        >;
!  partie Obs / probes
       if mobs.gt.0
       < do j=1,np
         < mult[i]=1,mobs : ct(mp+[.i],j) = f_d(Phi_tef(mp+[.i]))(/eta(j));   <<< Ct 2
         >;
       >;"End obs"
! partie parametres
! ----------------
       if lp.gt.0
       < ;mult[j]=1,lp: mult [i]=1,mp:
           Ct([.i],np+[.j]) = F_D(Phi_tef[i])(/eta(np+[.j]));
!  partie Obs
         if mobs.gt.0
         < ;mult[j]=1,lp: mult [i]=1,mobs:
           Ct(mp+[.i],np+[.j]) = F_D(Phi_tef(mp+[.i]))(/eta(np+[.j]));        <<< Ct 1
         >;
       >;
```

On a ainsi deux séries de colonnes à calculer, qui sont rangées dans les tableaux en extension du nombre de lignes de **Psi_t** et **Psi_p** :

– La partie sensible aux variables

```
! --- partie Obs : Psi_mu =  c2 Phi_t + d_fmu Psi_t
        if mobs.gt.0
        < call omatmlt(mobs,n,nxp,Ct(mp+1,1),mp+mobs,Phi_t(1,1),np  "c2 Phit"
                                             ,Psi_t(mp+1,1),mp+mobs);
          call omatmlt(mobs,n,nxp,D(mp+1,1),mp+mobs,Psi_t(1,1),mp+mobs
                                 ,Ctbuf,mobs);
          call omatadd(mobs,nxp,Ctbuf,mobs,Psi_t(mp+1,1),mp+mobs
                                             ,Psi_t(mp+1,1),mp+mobs);
        >;
```

```
        >;"End nxp"
```

- La partie sensible aux paramètres

```
  ! --- partie Obs : Psi_mu =  c1 + c2 Phi_p + d_fmu Psi_p

            if mobs.gt.0
            < call omatmlt(mobs,n,lp,Ct(mp+1,1),mp+mobs,Phi_p(1,1),np
                                                    ,Psi_p(mp+1,1),mp+mobs);
              call omatmlt(mobs,m,lp,D(mp+1,1),mp+mobs,Psi_p(1,1),mp+mobs
                                        ,Ctbuf,mobs);
              call omatadd(mobs,lp,Ctbuf,mobs,Psi_p(mp+1,1),mp+mobs
                                                    ,Psi_p(mp+1,1),mp+mobs);
              call omatadd(mobs,lp,Ct(mp+1,np+1),mp+mobs
                                 ,Psi_p(mp+1,1),mp+mobs,Psi_p(mp+1,1),mp+mobs);
            >;
        >;"End lp"
```

Enfin, une remise à zéro des sensibilités suit la même logique de déroulement pour les options **at** [**i**] **every** [**m**] ;.

Sensibilité des jacobiennes à un paramètre

Sur activation du selflag **dPI_aspha**, le code de **principal** calcule la sensibilité des Jacobiennes à un paramètre. Ces dérivées élementaires sont alors disponibles pour permettre différents calculs de sensibilité dans **ZSTEER**.

Il faut noter que la dérivation partielle par rapport au paramètre ne peut se faire pour une expression qui composerait un coefficient de matrice, masi qui n'est pas disponible – puisque celles-ci sont juste calculées numériquement. Il est donc nécessaire de repartir des formules du modèle (Deta_tef et Phi_tef) et d'en effectuer une double dérivation symbolique.

Pour fixer un objectif et comprendre l'utilité du code suivant, On donne l'exemple d'utilisation pour le calcul de la dérivée partielle de $A^\sharp$, avec les formules sont les suivantes :

$$\begin{aligned} d_\varpi A^\sharp = \quad & d_\varpi H + d_\varphi H\, d_\varpi \varphi \\ + \quad & \{d_\varphi B\, d_\varpi \varphi + d_\varpi B\}\,(I-D)^{-1} C^\dagger \\ + \quad & B\left\{(I-D)^{-1} d_\varpi C^\dagger - (I-D)^{-1} d_\varpi D\,(I-D)^{-1} C^\dagger\right\} \end{aligned} \tag{14}$$

Et pour suivre le code nécessaire, on donne l'exemple de dérivation double du calcul de la dérivée de H et B_b par rapport au paramètre (de symbole **po_asp** – on aussi besoin de la dérivée de φ :

```
!*KEEP,dPi_HBbF.
       ;mult [j]=1,mp : d_pi_ff[j]    = F_D(Phi_tef[j])(/po_asp);
       do j=1,np
       < mult[i]=1,np : d_pi_H([i],j) = F_D(F_D(deta_tef[i])(/eta(j)))(/po_asp);
       >;
       do j=1,mp
       < mult[i]=1,np :  d_pi_Bb([i],j)= F_D(F_D(deta_tef[i])(/ff(j)))(/po_asp);
       >;
```

Ces dérivées élémentaires sont combinées pour construire la sensibilité de aspha :

```
+SELF,IF=DPi_Aspha. **********************************************
! ---------------------
! Calcul de d_pi_aspha .
```

```
! -------------------------------------------------------------
!  D_p M = D_p[H] + D_f[H] D_p[f]
!                 + { D_f[B] D_p[f] + D_p[B] } (I-D)^ Ct
!                 + B { (I-D)^ D_p[Ct] - (I-D)^ D_p[D] (I-D)^ Ct };
! +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+SEQ,dPi_HBbF.       ================> d_p ff et celles de H et Bb, puis :
       call matcopy(d_pi_H,d_pi_aspha,n,np,n,np);
       call matmlt(d_pi_Bb,Ctbuf,To,n,np,m,mp,n,np);
       call matadd(d_pi_aspha,To,d_pi_aspha,n,np,n,np);
+SEQ,dPi_CtD.          ==============> dérivations doubles de Ct et D, suivies de :
       call matmlt(d_pi_D,Ctbuf,T_tmp1,m,mp,m,mp,n,np);
       call matsub(d_pi_Ct,T_tmp1,T_tmp2,m,mp,n,np);
       call matmlt(un_moins_d,T_tmp2,T_tmp1,m,mp,m,mp,n,np);
       call matmlt(Bb,T_tmp1,To,n,np,m,mp,n,np);
       call matadd(d_pi_aspha,To,d_pi_aspha,n,np,n,np);
       call veczero(T_tmp2,n*m);
```

Les calculs de la deuxième ligne (15) sont effectués en dérivant H et B_b par φ_k dans une boucle sur k :

```
   Do ko=1,m
   <
+SEQ,dffk_HBb. dérivation de H et Bb par rapport à ff(ko) :
     call scamat(d_pi_ff(ko),d_ffk_H,To,n,np,n,np);
     call matadd(d_pi_aspha,To,d_pi_aspha,n,np,n,np);
     call scamat(d_pi_ff(ko),d_ffk_Bb,T_tmp1,n,np,m,mp);
     call matadd(T_tmp2,T_tmp1,T_tmp2,n,np,m,mp);
   >;"End ko loop"

    call matmlt(T_tmp2,Ctbuf,To,n,np,m,mp,n,np);
    call matadd(d_pi_aspha,To,d_pi_aspha,n,np,n,np);
    if Zprint
    < title=' d_pi_aspha';
      CALL printmat(title,d_pi_aspha,n,np,n,np,0,0);
    >;
+SELF.
```

Ces calculs étant effectués dans le corps de principal, on n'a plus qu'à y ajouter son propre post-traitement, qui ici concerne ici comme illustration le calcul de la sensibilité des valeurs propres de $A^\sharp$ au paramètre, selon une idée de stb. Les calculs sont les suivants :

$$< f_i\, e_i > d_\varpi \lambda_i =< f_i \mid d_\varpi M \mid e_i > \tag{15}$$

avec f_i et e_i les couples de vecteurs propres à gauche et à droite de M. On a une formule équivalente pour les valeurs singulières .

Ceci se fait dans **ZSTEER**[1]. Noter au passage que la partie la plus laborieuse est ici le rangement des éléments propores déivrés par LaPack ! Le codage est de nature pédagogique, et ne recherche pas à faire court.

```
+SELF,IF=dPi_aspha.

!!============  CALCUL Valeurs et Vecteurs propres===============
```

[1]noter que le passage en double précision remplacera les **call SGEEVX** en **call DGEEVX** automatiquement par activation du **sel Double**.

```
!            <fi|ei> d_pi Li = <fi| d_pi_aspha |ei>
!! author Stephane BLANCO
!!====================================================================
      call matcopy(aspha,a1a,n,n,n,n);
      call sgeevx( 'N', 'V', 'V', 'N', n, a1a, n,
                     WR, WI, VL, n, VR, n, ILO, IHI, SCALE, ABNRM,
                     RCONDE, RCONDV, WORK,LoWORK,IoWORK,INFO );

5100; format (501(1x,e12.5));
5101; format (501(1x,e14.7));
      do jjk=1,n
      < tmp_cmp(jjk)=CMPLX(WR(jjk),WI(jjk));
      >;
     if (Zoutbef) < write(70,5101)time,tmp_cmp;>;

      do ival=1,n
      <
       IF (WI(ival).eq.zero)
         <
          do i_coord=1,n
          <
           vec_propre_l_reel(i_coord,ival)=VL(i_coord,ival);
           vec_propre_l_im(i_coord,ival)=zero;
           vec_propre_r_reel(i_coord,ival)=VR(i_coord,ival);
           vec_propre_r_im(i_coord,ival)=zero;
          >;
         >else
         <
          IF (WI(ival).gt.zero)
          <
          do i_coord=1,n
           <
           vec_propre_l_reel(i_coord,ival+1)=VL(i_coord,ival);
           vec_propre_l_im(i_coord,ival+1)=VL(i_coord,ival+1);
           vec_propre_r_reel(i_coord,ival)=VR(i_coord,ival);
           vec_propre_r_im(i_coord,ival)=VR(i_coord,ival+1);
           >;
          >else
          <
          do i_coord=1,n
           <
           vec_propre_l_reel(i_coord,ival-1)=VL(i_coord,ival-1);
           vec_propre_l_im(i_coord,ival-1)=-VL(i_coord,ival);
           vec_propre_r_reel(i_coord,ival)=VR(i_coord,ival-1);
           vec_propre_r_im(i_coord,ival)=-VR(i_coord,ival);
           >;
          >;
         >; "end IF (WI(ival).eq.zero"
      >; "end ival"

      do ival=1,n
      <
```

```
        do i_coord=1,n
        <
          comp_vec_propre_l(i_coord,ival)=
          CMPLX(vec_propre_l_reel(i_coord,ival),vec_propre_l_im(i_coord,ival));
          comp_vec_propre_r(i_coord,ival)=
          CMPLX(vec_propre_r_reel(i_coord,ival),vec_propre_r_im(i_coord,ival));
        >
       >
!!=============  CALCUL sensibilites valeurs propres ================!!
      do i_val=1,n
       <
         do i_coord=1,n
         <
          comp_vec_l(i_coord)=comp_vec_propre_l(i_coord,i_val);
          comp_vec_r(i_coord)=comp_vec_propre_r(i_coord,i_val);
         >;
         call vecvec_comp(comp_vec_l,comp_vec_r,comp_lr,n,n);
         call matvec_comp(d_aspha_dpi,comp_vec_r,comp_tmp_vec,n,n,n,n);
         call vecvec_comp(comp_vec_l,comp_tmp_vec,comp_lMr,n,n);

       tmp_cmp(i_val)=comp_lMr/comp_lr;
       >;
      if (Zoutbef)
      < write(71,5101)time,tmp_cmp;
      >;

+SELF.
```

Autres calculs dépendants des Jacobiennes

4.3.2 Intervention de mesures d'observables

Fonction critère et rencontre avec les points de mesure

Avance de la méthode de Newton pour la détermination de paramètres

Filtrage de Kalman

Processeurs peignes pour les modèles maillés

On illustre directement la méthode avec l'exemple suivant, tiré du modèle d'urbanysme de Gus. La fonction calcule l'écart quadratique de la variable **util2(.)** qui est un transfert[1]. On précise donc la variable du premier nœud et le nombre de mailles :

```
set_Probe
<  eqn: u_quad = vari_phi(n_node,util2(inode:1));
>;
```

Une macro transforme cette fonction en point d'entrée Fortran :

```
   ff(mp+1)=((VARI_NODE(n_node,ff(k2j_5(1)),j,FF(1),M_MULT)))
```

avec les arguments Fortran permettant à la routine de calculer les positions des variables dans le vecteur **ff(.)**. Ceci se fait en comptant M_MULT positions (ou le double en double precision) des adresses de taleaux.

La dérivation demande d'une part une macro :

[1] vari_eta(...) est la même fonction pour une variable d'état.

```
&'(VARI_NODE(#,#,#,FF(1),M_MULT))(/FF(#))'='(DVARI_NODE(#1,#2,#4,FF(1),M_MULT))'
```

avec appel au point d'entrée **dvari_node** qui, connaissant la vraiable de dérivation **ff(# 4)** fournit la dérivée. Remarquons l'absence de règle de dérivation (/eta(.)) qui résulte en une dérivée nulle.

Pour les fonctions avec plus d'arguments, une autre procédure est utilisée en déclarant explicitement le type des variables, exemple avec un calcul de moyenne pondérée :

```
! Moyenne ponderee
&'AVRG_POIDS(#,PHI:#,ETA:#)'='AVRG_TWO(#1,#2,#3,j,FF(1),ETA(1),M_MULT,N_MULT)'
&'AVRG_POIDS(#,ETA:#,PHI:#)'='AVRG_TWO(#1,#3,#2,j,FF(1),ETA(1),M_MULT,N_MULT)'
&'AVRG_POIDS(#,PHI:#,PHI:#)'='AVRG_TWO(#1,#2,#3,j,FF(1),FF(1),M_MULT,M_MULT)'
&'AVRG_POIDS(#,ETA:#,ETA:#)'='AVRG_TWO(#1,#2,#3,j,ETA(1),ETA(1),N_MULT,N_MULT)'
! poids type eta * ff
&'(AVRG_POIDS(#,PHI:#,ETA:#))(/FF(#))'
='(DAVRG_TWO1(#1,#2,#3,#4,FF(1),ETA(1),M_MULT,N_MULT))'
&'(AVRG_POIDS(#,ETA:#,PHI:#))(/FF(#))'
='(DAVRG_TWO1(#1,#3,#2,#4,FF(1),ETA(1),M_MULT,N_MULT))'
&'(AVRG_POIDS(#,PHI:#,ETA:#))(/ETA(#))'
='(DAVRG_TWO2(#1,#2,#3,#4,FF(1),ETA(1),M_MULT,N_MULT))'
&'(AVRG_POIDS(#,ETA:#,PHI:#))(/ETA(#))'
='(DAVRG_TWO2(#1,#3,#2,#4,FF(1),ETA(1),M_MULT,N_MULT))'
! poids type ff * ff
&'(AVRG_POIDS(#,PHI:#,PHI:#))(/FF(#))'
='(DAVRG_TWO1(#1,#2,#3,#4,FF(1),FF(1),M_MULT,M_MULT)
+DAVRG_TWO2(#1,#2,#3,#4,FF(1),FF(1),M_MULT,M_MULT))'
&'(AVRG_POIDS(#,ETA:#,ETA:#))(/ETA(#))'
='(DAVRG_TWO1(#1,#2,#3,#4,ETA(1),ETA(1),N_MULT,N_MULT)
+DAVRG_TWO2(#1,#2,#3,#4,ETA(1),ETA(1),N_MULT,N_MULT))'
```

La récupération des variables et leur dérivées suivent la même logique que dans le cas plus simple.

Au 27/02/07, on a en stock les fonctions de calcul de moyenne simple ou pondérée, de variance en accord, et un exemple de fonction ad'hoc calculant un indice de Gini, avec dérivées par accroissements finis.

4.4 Calcul de l'adjoint

On obtient le système adjoint à partir de (7)

$$\begin{bmatrix} -\partial_t v_\varpi \\ -\partial_t v \\ w \\ w_\mu \end{bmatrix} = \begin{bmatrix} 0 & H_\varpi^\dagger & C_\varpi & c_1^\dagger \\ 0 & H^\dagger & C & c_2^\dagger \\ 0 & B_b^\dagger & D^\dagger & d_{\varphi\mu}^\dagger \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_\varpi \\ v \\ w \\ w_\mu \end{bmatrix} + \begin{bmatrix} l_\varpi \\ l_\eta \\ l_\varphi \\ l_\mu \end{bmatrix} \tag{16}$$

système dans lequel $v_\varpi(\tau)$ représente la sensibilité (indirecte) de la fonction de coût $J(T)$ à une modification des paramètres (à $t = \tau < T$), et ainsi des autres variables adjointes. Les sources l_x sont donnés par les dérivées partielles de l'intégrande **cout_l** (sensibilités directes).

Bien sùr, la sensibilité (indirecte) w_μ aux probes est découplée du système, puisque les autres variables n'en dépendent pas par construction ; par contre, J pourra explicitement être fonction de ces observables ($l_\mu = \partial_\mu \ \{(\mu(t_i) - M_i)^2\}$ dans le cas classique de la minimisation de l'écart quadratique obs-mesures).

Bref, on a en gros une situation où à nouveau le système couplé est le même que sans paramètres ni probes, et des simplifications de calcul de ces composantes.

Trajectoire en arrière

L'intégration à la TEF du système (16) donne

$$\begin{bmatrix} -I - \frac{dt}{2}H^\dagger & -\frac{dt}{2}C \\ -B_b^\dagger & I - D^\dagger \end{bmatrix} \begin{bmatrix} \delta v \\ \delta w \end{bmatrix} = \begin{bmatrix} \Gamma_v \\ \Omega_w \end{bmatrix} + \begin{bmatrix} l_\eta \delta t \\ l_\varphi \end{bmatrix} \tag{17}$$

pour la partie couplée du système (cf (23) du cahier Adjoint), avec $\Omega_w = 0$, et ou la contribution explicite est $\Gamma_v = \delta t[H^\dagger v + Cw]$.

La première étape consiste à récupérer les Jacobiennes. Le principe d'économie consiste en un compromis : on sauve la trajectoire avec à la fois les vecteurs η et φ (mais pas φ_ϖ), pour éviter la D-loop (A). On récupère également la commande lorsqu'une loi est calculée en avant (B).

Les Jacobiennes sont calculées classiquement (C), nous verrons le calcul concernant la commande, les paramètres et les probes ultérieurement.

On passe des Jacobiennes aux matrices adjointes du TEF par transposition (**call otransmat**, (D)) et avec $A^\dagger = -I - \frac{dt}{2}H^\dagger$ et c'est la $C = C^{\dagger\dagger}$ qui est ici multipliée par $-\frac{dt}{2}$:

```
! **************  Restitution fin pas retour **************

      time=time_sauve(istep);
      dt = time-timebef;
      <i=1,np; eta(i) = eta_sauve(i,istep); >;                (A)
      <j=1,mp; ff(j) = ff_sauve(j,istep); >;
;
      if Zcommand
      < <i=1,np; ux_com(i) = ux_sauve(i,istep); >;            (B)
        <j=1,mp; uy_com(j) = uy_sauve(j,istep); >;
      >;

if ZPRINT< z_pr:timebef,time,dt,eta; z_pr:ff; z_pr:cout_J; >;

! Fonction de cout  (dt<0) par trapèzes
```

```
  coutstep = - (coutlbef+cout_l)*0.5*dt;
  cout_J = cout_J + coutstep;
;
! *************   Appel de la resolution  ***************
! ---------------- ( time backward ) ---------------------
+SEQ,D_Mx.
+SEQ,Matrice_ABCt.                                          (C)
! ====== calculs dependants de dt
! build adj a mx(init a I) from h : a = -I -dt/2*h
      call oscamat(-dt/2.,n,n+lp,h,np,hdt2m,np);
      call diagmat(-one,a,n,np+lp);
      call omatadd(n,n+lp,a,np,hdt2m,np,a,np);
! Build adj B mx from Bd  (identique)
      call omatcopy(n,m,Bb,np,B,np);

+SEQ,trans_abcd. et calculs vecteurs                        (D)

! ----------------------------------------------------------
  CALL oker("in" n,m,0,a_c,np+lp,ct_c,np+lp,b_c,mp,d_c,mp,
                 gam_adj,np,omeg_adj,mp,v_adj,np,w_adj,mp,    (E)
           lp,mobs,EtaNam,PhiNam,ParNam,zprint,
          "out" dv_adj,dphi, "in:" 0, "out:"gain,infoa,infores);
! ----------------------------------------------------------
! No error report as there were no error during the direct run
;
! incrementations
! -----------------------------------------------           (F)
  <i=1,np+lp;  v_adj(i) = v_adj(i) + dv_adj(i); >;
   coutlbef = cout_l;
   timebef = time;
   ZPRINT = mod(istep-1,modzprint).eq.0 .or.istep.eq.2
                                        .or.istep.eq.nstep;
   zprint=zprint.and.zprcall;
   Zout = mod(istep-1,modzout).eq.0;
   Zout = Zout .or. istep.eq.1 .or.(istep+1).eq.1;

  >;"end time loop"
! ------------------------------------------------------ end back loop
```

La résolution se fait par appel de **oker** (E) avec échange des matrices transposées par rapport aux calculs en avant, et on incrémente l'état adjoint (ici avec **nzp** nul, c'est-à-dire sans les extensions du direct, qui consisteraient en un calcul de plusieurs fonctions de coût à la fois[1]).

Le parcourt de la trajectoire s'effectue à rebrousse-temps avec **dt** négatif :

```
!------------------------------------------------------ back time loop
  DO istep = nstep,0,-1
  <
    IF ZPRINT< print9000; print *,'back_istep',istep,time; print9000; >;

! ----------------------------------------------------------
```

[1]extension possible mais non encore envisagée.

```
!  ********** calcul du w debut de istep avec D et B bef **
!             ( I-Dt) w = Bt v + l_y
! ----------------------------[sauf a l'entree]-----------
```

avec pour le pas de temps initial utilisation des matrices $D^\dagger$ et $Bb^\dagger$ du dernier pas en avant – alors que le calcul en avant devait commencer par calculer la D-loop et $C^\dagger$. La source en provenance de l'intégrande de J ne contribue qu'à partir du pas de temps suivant, c'est-à-dire le premier pas de l'intégration.

Comme en avant, on commence par effectuer les calculs de la fin d'un incrément (ou des valeurs initiales) :

```
      CALL matvec(b_c,v_adj,b_c_v_adj,m,mp,n,np);

! ------------------------------------------------
! fin calcul w fin bef : ajout l_y et resolution
! ------------------------------------------------

    CALL matvec(un_moins_d_c,b_c_v_adj,w_adj,m,mp,m,mp);
;
    >;"end if istep courant"

! **************  Ecriture des valeurs a timebef **********
             o o o

      if istep.eq.0 < goto :exitB:; >;
;
```

Sensibilité aux paramètres et vecteurs connus

Comme en avant, la sensibilité aux paramètres de la **free_parameters** liste est explicite une fois connue l'avance des autres (cf 16) :

$$-\delta v_\varpi = \delta t \left[H^\dagger_\varpi v + Cw + l_\varpi \right] + \frac{dt}{2} H^\dagger_\varpi \delta v + \frac{dt}{2} C_\varpi w \tag{18}$$

la seule difficulté réside ainsi dans la prise des bons signes et conventions, qui ont été choisies pour l'appel à **ker**, avec le résultat :

```
! partie parametres : calcul direct de p_adj / v_adj et w_adj
! ------------------------------------------------------------
! d p_adj = A' dv_adj + C' dw_adj - gam_adj'   (a1a used as working space)
      if lp.gt.0
      < nw=np+1;
        call matvec(a_c(nw,1),dv_adj,dv_adj(nw),lp,np+lp,n,np);
        call matvec(ct_c(nw,1),dphi,a1a,lp,np+lp,m,mp);
        call vecadd(dv_adj(nw),a1a,dv_adj(nw),lp);
        call vecsub(dv_adj(nw),gam_adj(nw),dv_adj(nw),lp);
      >;
;
! ------------------------------------------------
! incrementations
! ------------------------------------------------
  <i=1,np+lp;  v_adj(i) = v_adj(i) + dv_adj(i); >;
```

Enfin, les vecteurs connus (genre Γ, Ω) sont auparavant déterminés classiquement, ce qui fait que nous regroupons ici ces calculs :

```
! calcul des vecteurs Gam_adj et omeg_adj
! Gam = dt  ( l_x + Ht v + C w  )
! ----------------------------------------
     CALL matvec(H_c,v_adj,gam_adj1,n,np+lp,n,np);            (A)
     CALL matvec(ct_c,w_adj,gam_adj2,n,np+lp,m,mp);
     CALL vecadd(gam_adj1,gam_adj2,gam_adj,n);
! Omeg = 0
     <j=1,mp; omeg_adj(j) = 0.; >;

! ajout source l_x                                            (B)
     <i=1,n; gam_adj(i) = gam_adj(i) + f_d(cout_l)(/eta(i)); >

! partie parametres :
! gam_pi_adj = dt  ( l'_x + Ht' v + C' w  )
! ----------------------------------------                    (C)
     if lp.ne.0
     < nw=np+1;
       CALL matvec(H_c(nw,1),v_adj,gam_adj1(nw),lp,np+lp,n,np);
       CALL matvec(ct_c(nw,1),w_adj,gam_adj2(nw),lp,np+lp,m,mp);
       CALL vecadd(gam_adj1(nw),gam_adj2(nw),gam_adj(nw),lp);
!    addition de l'x
       ;mult[i]=1,lp : gam_adj(np+[.i]) =
                       gam_adj(np+[.i]) + f_d(cout_l)(/eta(np+[.i]));
"---">;"end parametres ------------------"
;
     CALL scavec(dt,gam_adj,gam_adj,n+lp);                    (D)
```

On a successivement :

- la partie linéaire parallèle à $H\eta + B_b\varphi$ (A) ;
- à laquelle on ajoute la contribution explicite de J : $\partial_\eta J$ (B) ;
- les paramètres ont leur incidence sur v_{adj} et w_{adj} données par l'extensions aux matrices (C) ;
- et l'intervention du pas de temps (D) ;

Sensibilité à la commande

La grande différence avec le direct est le calcul de sensibilité de J à la commande, qui est traitée à part, cf formule (19) (la 21 du cahier Adjoint), alors que l'on aurait pu la considérer comme étant spécifiée par des composantes de transferts – leur différence ici étant le choix d'absence de matrice D et la simplicité de leur application (une loi distincte par composante d'état et de transfert) :

$$\begin{aligned} \nabla_{ux} J(\tau,T) &= \; < v(\tau) \mid B^u(\tau) \quad + < l_{ux}(\eta(\tau), h^x(\tau)) \mid \\ \nabla_{uy} J(\tau,T) &= \; < w(\tau) \mid D^u(\tau) \quad + < l_{uy}(\eta(\tau), h^y(\tau)) \mid \end{aligned} \tag{19}$$

On a donc incidence de Jacobiennes du modèle par rapport aux commandes à déterminer, d'où la programmation avec la partie commande via états (E) suivie de commande via transferts (F) :

```
!  calcul de Grad_ue J = <vB'| + l_ux
!  calcul de Grad_uf J = <wD'| + l_uy :
! ------------------------------------
      if Zcommand
      <                                                       (E)
```

```
        ;mult[i]=1,np: GradueJ[i] = f_d( v_adj[i]*(deta_tef[i])
                                     + cout_l )(/ux_com[i]);
        ;mult[j]=1,mp: GradufJ[j] = w_adj[j]*f_d(Phi_tef[j])(/uy_com[j])
                                + f_d(cout_l)(/uy_com[j]);
      >;                                                        (F)
;
    >;"end if istep courant"
! **************  Ecriture des valeurs a timebef **********
```

Il reste l'imputation de la fonction de coût terminale **coût-Psi** que l'on calcule donc en tête de l'adjoint (le code se trouve dans la séquence **Cout_Psi**)

```
!       Initialisation v(T) -> pour zinit-adjoint
! ----------------------------------------------------------------
!  |v(T)> = |d_eta cout_Psi> + Sum_j d_ff(j) cout_Psi |C(I-D')-1>_j
! ----------------------------------------------------------------
! - partie dependance de eta
  <i=1,n; v_adj(i) = f_d(cout_Psi)(/eta(i)) >;                  (G)

! - partie dependance de ff (Ctbuf=CtDm1(T) est disponible)
  do j=1,m
  < aaa = f_d(cout_Psi)(/ff(j));
    if abs(aaa).ge.1.E-06
    < <i=1,n; v_adj(i) = v_adj(i) + aaa*Ctbuf(j,i) >;           (H)
    >;
  >;
  if abs(aaa).ge.1.E-06 <Print*,' *** cout_Psi dependant de ff(T) pulse ***';>;

! - partie parametres
  if lp.gt.0                                                    (I)
  < ;mult[i]=1,lp : v_adj(np+[.i]) = f_d(cout_Psi)(/eta(np+[.i]));
  >;
```

On remarque en plus de la contribution de l'état final (G) une éventuelle contribution de la valeur d'un tranfert final (H) – considéré alors comme générant un pulse sur l'état comme il est à présent admis (cf cahier Effluents).

Enfin des paramètres pouvant apparaître explicitement dans le finale, on a ce terme (I) – dans lequel rappelons que **eta(np+[.i])** sera remplacé tour à tour (**mult[i]**) par Mortran par chaque symbole des paramètres de la **Free** liste.

Traitement des probes

En suivant cette fois l'ordre séquentiel, on calcule d'abord w_μ :

```
}
!  ********** calcul du w_mu debut de istep avec D et B bef **
!              w_mu =  l_mu
! --------------------------------------------------------
    if mobs.gt.0
    < if istep .lt. nstep
      < <j=1,mobs; w_adj(mp+j) = f_d(cout_l)(/ff(mp+j)); >;
      >else
      < <j=1,mobs; w_adj(mp+j) = f_d(cout_Psi)(/ff(mp+j)); >;
      >;
    >;
```

qui fournit un term additionnel (J) au calcul de w à la fin du pas de temps :

```
}
! --------------------------------------------------------
!  ********** calcul du w debut de istep avec D et B bef **
!            ( I-Dt) w = Bt v + d'_ffmu w_mu + l_y
!                                    (J)

      CALL matvec(b_c,v_adj,b_c_v_adj,m,mp,n,np);
      CALL matvec(d_c(1,mp+1),w_adj(mp+1),d_c_w_mu,m,mp,mobs,mobs);  (J)

    if mobs.gt.0
    < <j=1,mp; b_c_v_adj(j) = b_c_v_adj(j) + d_c_w_mu(j); >;         (J)
    >;

    if istep .lt. nstep
    < <j=1,mp; b_c_v_adj(j) = b_c_v_adj(j) + f_d(cout_l)(/ff(j)); >;
    >else
    < <j=1,mp; b_c_v_adj(j) = b_c_v_adj(j) + f_d(cout_Psi)(/ff(j)); >;
    >;

    CALL matvec(un_moins_d_c,b_c_v_adj,w_adj,m,mp,m,mp);
```

On a aussi des contributions au calcul des Γ (K) :

```
}
! ---------------------------------------------------
! calcul des vecteurs Gam_adj (et omeg_adj  nul)
! Gam_adj    = dt ( l_x + Ht v + C w + c_2 w_mu )
! Gam_pi_adj = dt ( l_pi + Ht' v + C' w + c_1 w_pi )
! ---------------------------------------------------

     CALL matvec(H_c,v_adj,gam_adj1,n+lp,np+lp,n,np);
     CALL matvec(ct_c,w_adj,gam_adj2,n+lp,np+lp,m+mobs,mp+mpbs);     (K)
     CALL vecadd(gam_adj1,gam_adj2,gam_adj,n);

     <j=1,mp; omeg_adj(j) = 0.; >;

! ajout sources l_x, l_pi
     <i=1,n; gam_adj(i) = gam_adj(i) + f_d(cout_l)(/eta(i)); >
     if lp.gt.0
     < ;mult[i]=1,lp : gam_adj(np+[.i]) =                            (K)
                       gam_adj(np+[.i]) + f_d(cout_l)(/eta(np+[.i]));
     >;
```

Calcul du Hamiltonien

Toujours dans le cahiers Adjoint, on a une formulation de la fonction dîte du Hamiltonien qui doit être calculée à chaque instant :

$$H(\eta, \varphi, v, w, h^x, h^y) = <\ v\ g(\eta, \varphi, h^x) > + <\ w\ f(\eta, \varphi, h^y) > + l(\eta, \varphi, h) \tag{20}$$

son intérêt rappelons le concernant son extrêmalité en u considérée comme simple variavle d'une part, et de l'autre lors de la recherche d'optimalité du temps terminal ($H(T)$ devant alors s'annuler.

```
!  ********** Fonctions dependant de <v| et <w| (t*********
! -----------------------------[sauf a l'entree]-----------
    if istep .lt. nstep
    <
! -*********************************
!   Calcul du Hamiltonien courant :
! -*********************************
!  H(eta,phi,v,w,ux,uy) =
!                  <v| g(eta,ff,ux)> + <w| f(eta,ff,uy) + l(eta,phi,u)
      Hamilton = 0.;
      mult [i]=1,np : Hamilton = Hamilton + v_adj[i]*(deta_tef[i]);
      mult [j]=1,mp : Hamilton = Hamilton + w_adj[j]*(Phi_tef[j]);
      Hamilton = Hamilton + cout_l;
```

la variable **cout_l** étant l'intégrande de J donnée dans **Zinit** par une **f_set**.

Variables et fichiers de l'adjoint

nom	dim 1		*signification*	data file
v_adj	np	1	vecteur adjoint d'état	vadj.data
v_adj	+lp	1	vecteur adjoint paramètres	gradpj.data
dv_adj	np+lp	1	avance	no data
GradueJ	np	1	sensibilités à la commande d'état	graduxj.data
w_adj	mp	1	vecteur adjoint de transfert	wadj.data
w_adj(+)	+mobs	1	partie observables	gradmuj.data
GradufJ	mp	1	sensibilités à la commande de transferts	graduyj.data

4.5 Recherche d'extremum de fonctionnelle

5 de quelques analyses dynamiques

5.1 Vecteurs de Floquet

L'idée serait ici, un brouillon avant codage, d'avoir une cohérence entre le calcul des Floquets et des gains et réponses du système périodique. Les algorithmes suivants sont pensés pour leur passage à MK2 et à sa navette.

recherche d'agorithme efficace On part des remarques des **Effluents**, on prolonge jusqu'à l'explicite. L'idée est d'éviter le plus possible les variables complexes, en profitant de leur rangement par couple de colonnes $\{| x_r >, | x_i >\}$ pour tout ce qui leur applique des produits de matrices réelles. On a alors une partie du calcul qui avance comme les sensibilités, et on post-traite pour appliquer les corrections dûes aux produits complexes. *Dans le exos de Céline et Pablo, on s'embêtait pas et on calculait* $\Psi^F(t) = \Phi(t,0)Zexp(-\Lambda t)$ *en complexes ; mais d'une part, on avance les deux parties de chaque paire, l'imag nulle des réels etc. Par ailleurs, du fait de la précision numérique, on obtenait des V.F. erratiques sur lesquels on fermait les yeux — garbage out...*

On suppose que $\Psi^r(t) = \Psi^F(t)$ — c'est-à-dire formées des mêmes colonnes — et $\Psi^r(t+\tau) = \Phi(t+\tau,t)\Psi^r(t)$, ce calcul de $\delta\Psi^r$ séparant les deux étant effectué dans **oker**. On a alors

$$
\begin{aligned}
\text{à } t: \ \Phi^r &= \Phi^F \rightarrow \text{MK} \rightarrow \Phi^r(t+\tau) = \Phi(t+\tau,t)\Phi^F \\
&\quad \text{d'où : } \delta\Phi^r \ ; \\
\Phi^F(t+\tau) &= \Phi(t+\tau,t)\Phi(t,0)Ze^{-\Lambda(t+\tau)} \\
&= \Phi(t+\tau,t)\Phi^F(t)e^{-\Lambda\tau} \\
&= \Phi^r(t)\ e^{-\Lambda\tau}
\end{aligned}
$$

d'où il suit que

$$\boxed{\Phi^F = \Phi^r \exp(-\Lambda\tau).} \tag{21}$$

On a ainsi une correction avec matrice diagonale de coeffs complexes ou non. On a autant de colonnes à avancer que celles de Φ_t, d'où l'idee de l'initialiser avec les bonnes parties de Z, l'avancer classiquement et appliquer (21) vers ZSTEER. Il y a ainsi à construire une gestion des colonnes et corrections, dont la mise au point suit ce paragraphe.

Algorithme de construction des pointeurs et tableau des v.p. On part des sorties de LaPack au moment du stockage de Z à la première période. On suppose pour fixer les idées $np = 7$, avec une suite de vp réelle, complexe, réelle, complexe, réelle. Comme remarque supplémentaire, on doit tenir compte de l'existence possible d'un multiplicateur négatif, correspondant à un V.F. réel corrigé par un facteur de type $\cos(\omega t)exp(-\lambda t$, c'est pourquoi on conserve sous forme complexe tous les exposants de Floquets pour intégredr ce cas à la correction du cas réel.

```
+SELF,IF=Flo_vec.
      complex Flo_Z(np,np),Flo_sigma(np),ovpF(np),wox;
      real Flo_Period,aor,bor,orvec(np),oivec(np),ww,dtby2;
      integer ko_colF(2*np);

+SELF,IF=Flo_vec.
! -----------------------------------------------------
!               r + - r + - r    (rgt en sortie de Lapack)
!   exemple WR : * * * * * * *    n=7
!           WI : 0 *-* 0 *-* 0        => =2n (Lapack donne toutes les paires)
```

```
!                1 2 3 4 5 6 7
            1  2  3  4  5  6  7
!    ovPF  : *0,**,00,*0,**,00,*0 Cx
!
!  rgt des v.p r+is                          Il doit etre de meme dim que Phi_F
!        vpF  : r 0 r i 0 r i r 0   = 2n    mais en n*cmplx
!  ptr colonnes de Phi^F et yes/no correction   ************
!       kcolF : 1,no 2,yes 4,no 5,yes 7,no  <= 2n  (ici 10)
! ! ----------------------------------------------------------
! ozvf dit r1 pour colonne relle, R2,I2 pour complexe etc
character*2 ozvf(np);
;
! calcul des exposants de Floquet
! -------------------------------
OPEN (50, File='flovp.init', status='OLD');
      READ (50,1009) Flo_Period, (Flo_sigma(i),i=1,np);
      CLOSE(50);
;
! et initialisation de Phi_t avec Z
! ---------------------------------
OPEN (50, File='floVP.init', status='OLD');
      READ (50,1009) ww, ((Flo_Z(i,j),i=1,np),j=1,np);
      CLOSE(50);
;
  <i=1,2*n; ko_colF(i)=0; ovpF(i)=(0.,0.) >;
   i=1;"ptr sur une premiere v.p. des paires"
   k=1;"info VP (une info par paire)"
  While i.le.n
  < ovpF(i)= log(Flo_Sigma(i))/Flo_Period;
!   (Rq : le 2eme ovpF d une paire reste nul)
    if Imag(Flo_Sigma(i)).eq.0
    < ko_colF(k)=i; ko_colF(k+1)=-1"no";
      ozvf(i)='r'//charit2(i);
      <j=1,n;Phi_t(j,i)=Real(Flo_Z(j,i));>;
     [i:+1];
    >else
    < ko_colF(k)=i; ko_colF(k+1)=1"yes";
      ozvf(i)='R'//charit2(i); ozvf(i+1)='I'//charit2(i);
      <j=1,n;Phi_t(j,i)=Real(Flo_Z(j,i));Phi_t(j,i+1)=Imag(Flo_Z(j,i));>;
     [i:+2];
    >;
   [k:+2]
  >;"EndDo"
  print9000;
  Z_pr/Floquet/: ((j,Phi_t(i,j),i=1,np),j=1,np);
  Z_pr: Flo_period, Flo_Sigma;
  Z_pr/mult de Floquet/: ovpF;
```

Calculs correctifs On ne fait évidemment avancer qu'un seul Floquet complexe de chaque paire. À faire vers ZStep (on utilise Phi(t)) :

```
!##################
```

```
+SELF, IF=FLO_VEC.
!##################
! Epreuve d'algorithme a rentrer dans le code standard
! ---------------------------------------------------------------
! Phi_t = Phi_t exp(-Lambda.dt) ou Phi_t(t) exp(-Lambda. t)
! à choisir en fonction des multiplicateurs.
! Version integrant un multiplicateur reel negatif
! ---------------------------------------------------------------
  Do io=1,2*n,2
  <
   ko=ko_colF(io); "ptr 1st col"
!    Calcul du facteur correctif
!    --------------------------
   wox = exp(-ovpF(ko)*dt);
   aor = real(wox); bor = imag(wox);
    if ko_colF(io+1).eq.1                    "cas complexe"
    <
!    Correction du cas complexe
!    -------------------------
     call omatcopy(n,1,Phi_t(1,ko),np,orvec,np);
     call omatcopy(n,1,Phi_t(1,ko+1),np,oivec,np);
     Do jo=1,np
     < Phi_t(jo,ko)   = aor*orvec(jo) - bor*oivec(jo);
       Phi_t(jo,ko+1) = bor*orvec(jo) + aor*oivec(jo);
     >;"EndDo"
!    ---------------------------
!     Correction du cas reel ----
!    ---------------------------
    >elseif ko_colF(io+1).eq.-1                        "cas reel"
    <
      Do jo=1,np
      < Phi_t(jo,ko) = aor*Phi_t(jo,ko);
      >;"EndDo"
;
    >;"EndIf"
;
  >;"end Do"
```

On devrait alors avoir successivement

1. à la détermination des Floquets : stockage Z et σ en complexe (dimension np) ;
2. init : calcul des multiplicateurs de Floquet ; et rangement kCol(1 :¡2n) et vpF (dim np) ;
3. initialisation de Phi_t avec Z ;
4. à chaque fin du pas de calcul : corrections d'avance complexe ou réelle ;
5. (orvec et oivec sont deux vecteurs colonnes de dim np) ;

avec cette solution, rien à changer dans les calculs d'avance. Essais validés avec ClimSI, puis avec Evex (en double) qui a des Floquet complexes.

Analyse de Floquet de l'Effet-en-retour ici pour l'instant :
pour la partie avance d'une période

```
! Calcul de Phi_t et Psi_t en avant (nxp=np)
! -------------------------------------------
 Sensy_to_var
 <         var: T_ocean_eta   , pert: init;
           var: T_tropo_eta   , pert: init;
           var: T_strato_eta  , pert: init;
           var: Q_tropo_eta   , pert: init;
           var: CO2_conc_eta   , pert: init;
! Gain de rétroaction
           var: T_WV_phi      , pert: step_eff;
 >;
! moderateur sensib a xCO2 :
!***  Phi_t(5,5)=3.30d-04; "XXX"
\end{dverbatim}
%
pour la partie avance d'une période (II) varrho_m
%
\begin{verbatim}
! Calcul de Phi_t et Psi_t en avant (nxp=np)
! -------------------------------------------
 Sensy_to_Eta
 < "avance de sigma^F : pour chaque rep-eff"
!               il faut en prendre une et initialiser la colonne avec sig_T
           var: T_ocean_eta   , pert: init;   on a un pb de notation alors ?
           var: T_tropo_eta   , pert: init;
           var: T_strato_eta  , pert: init;
           var: Q_tropo_eta   , pert: init;
           var: CO2_conc_eta  , pert: init;  celle
 >;
! Rep-eff rétroaction
 Sensy_to_Phi
 <           var: T_WV_phi      , pert: step_eff;
 >;
! moderateur sensib a xCO2 :
!***  Phi_t(5,5)=3.30d-04; "XXX" à présent dans minikq.car :
       step_psi(6)=3.30d-04;

! copie de sigma(T) dans les colonnes de Phi_t
! copie de Z dans Phi_t nxn
```

Il faudrait une notation pour reserver la place des Floquet — et pour σ^F ?

En profiter pour ajouter une source symbolique attachée à une rep-eff ? Il faudrait lister les sources dans une ko_p_source(k)=nom de la fonction définie par fun_set source=f(t), calculer ces sources par une mult[] et les attacher à step_psi(ko)=source ;

Finalement, hors flo-resp, on a nxp total, nyp=nyp2=0 ou le nbre des phi sensy ; et pour flo-resp, nxp total, nyp2 de phi-sensy, nyp=2*nyp (on doit avoir np=nxp-nyp).

Les macros 102 de sensy-to-var corrigées pour Floquet-effets-en-retour :

```
! ---------------------------------------------------------------------------
! Sensitivity definition
! Le principe des defauts est que l'instruction
!  if (mod(istep+1,ko_p_mod(1)).eq.ko_p_ini(1)+1) Zko_p(1)=.true.;
```

```
! done un toujours .false. [et mod(x,-1) donne tjrs 0 ]
&'$PERT:INIT;'='ko_p_typ(#C-1)=1;' &'$PERT:STEP_EFF;'='ko_p_typ(#C-1)=4;'
&'$PERT:STEP_RESP;'='ko_p_typ(#C-1)=3;' &'$PERT:PULSE;'='ko_p_typ(#C-1)=2;'
&'$AT#;'='ko_p_ini(#C-1)=#1;ko_p_mod(#C-1)=-1;'
&'$AT#EVERY#;'='ko_p_ini(#C-1)=#1;ko_p_mod(#C-1)=#2;'
&'ETA(#)$VT$'='ko_p_tvar(#C)=1;ko_p_kvar(#C)=#1;'
&'FF(#)$VT$'=' ko_p_tvar(#C)=2;ko_p_kvar(#C)=#1;'
&'$PRT:INIT;'='ko_p_typ(nxp-nyp+#C-1)=1;'
&'$PRT:STEP_EFF;'='ko_p_typ(nxp-nyp+#C-1)=4;'
&'$PRT:STEP_RESP;'='ko_p_typ(nxp-nyp+#C-1)=3;'
&'$PRT:PULSE;'='ko_p_typ(nxp-nyp+#C-1)=2;'
&'$AS#;'='ko_p_ini(nxp-nyp+#C-1)=#1;ko_p_mod(nxp-nyp+#C-1)=-1;'
&'$AS#EVERY#;'='ko_p_ini(nxp-nyp+#C-1)=#1;ko_p_mod(nxp-nyp+#C-1)=#2;'
&'ETA(#)$VTF$'='ko_p_tvar(nxp-nyp+#C)=1;ko_p_kvar(nxp-nyp+#C)=#1;'
&'FF(#)$VTF$'=' ko_p_tvar(nxp-nyp+#C)=2;ko_p_kvar(nxp-nyp+#C)=#1;'
! on pourrait compter les etats et transferts et remplir nyp (a voir avec l unsecte)
&';SENSY_TO_VAR<'='Call veczero(PHI_T(1,1),np*nxp);WRITE(*,*);
WRITE(*,*)'' --------------- Informing on Sensitivity to variables ----------'';
;WRITE(*,*)''  Var-name,              Perturbation-type,    column in Phit Mx'';
#L8#S1#L #S6#N; &''VAR:##,''=''##U6##S6;##1$VT$;,##1$''
 &'',##$PERT:##;''='' ko_p_ini(##C)=1;ko_p_mod(##C)=-1;
 WRITE(*,''''(2A24,I3)'''')''''##1'''',''''##2'''',##U6##C;$PERT:##2;##A1##S6;''
 &'',##$PERT:## AT''=''
 WRITE(*,''''(2A24,I3)'''')''''##1'''',''''##2'''',##U6##C;$PERT:##2;##A1##S6;
$AT'' &''$8$''=''##U6;##N;
 WRITE(*,*)'''' ----------------------------------------------------'''';
 IF(nxp.NE.##C-1)
WRITE(*,*)''''**WARNING : Mismatched definition / parameter nxp='''' ,nxp,##C-1;
 &''''####K$8$'''' &''''####KVAR:####,'''' &''''####K,####$PERT:####;''''
jo=0;ko=0;
DO io=1,nxp;IF (ko_p_typ(io).eq.1.and.ko_p_tvar(io).eq.1.and.ko_p_ini(io).eq.1)
THEN;Phi_t(ko_p_kvar(io),io)=1.;ko=ko+1;ENDIF;
IF (ko_p_tvar(io).eq.1.and.ko_p_typ(io).ne.1)THEN;
write(*,*)''''***ERROR : Cannot set Pert_type'''',ko_p_typ(io),
'''' on State variable:'''',EtaNam(ko_p_kvar(io));jo=1;ENDIF;
IF (ko_p_tvar(io).ne.1.and.ko_p_typ(io).eq.1)THEN;
write(*,*)''''***ERROR : Cannot set C.I. sensy type for Transfer variable:''''
,PhiNam(ko_p_kvar(io));jo=1;ENDIF;ENDDO;
+SELF,IF=Flo_resp.  ----------------------------------------------------
IF(ko.NE.N)THEN;WRITE(*,*)''''***ERROR all State Variable required in sensy
to Var block'''';jo=1;
! copie de Z dans Phit et de sigmaT dans son double
;####* === Initialisations: Z et sigma_T dans PhiT(1:n,1:2n) ============;;
! *** A FAIRE ***
+SELF.
IF (jo.ne.0)write(loerr,*)'''' *** ERROR in ZINIT Sensy_to_var'''',char(7);'''
! ------------------- new commands added --------------------------------------
&';SENSY_TO_ETA<'=';WRITE(*,*);
WRITE(*,*)'' --------------- Informing on Sensitivity to Eta-variables ------'';
;WRITE(*,*)''  Var-name,              Perturbation-type,    column in Phit Mx'';
#L8#S1#L #S6#N; &''VAR:##,''=''##U6##S6;##1$VT$;,##1$''
 &'',##$PERT:##;''='' ko_p_ini(##C)=1;ko_p_mod(##C)=-1;
```

```
 WRITE(*,''''(2A24,I3)'''')''''##1'''',''''##2'''',##U6##C;$PERT:##2;##A1##S6;''
 &'',##$PERT:## AT''=''
 WRITE(*,''''(2A24,I3)'''')''''##1'''',''''##2'''',##U6##C;$PERT:##2;##A1##S6;
$AT'' &''$8$''=''##U6;##N;
 WRITE(*,*)'''' ------------------------------------------------------'''';
 &''''####K$8$'''' &''''####KVAR:####,'''' &''''####K,####$PERT:####;'''' jo=0;
Call veczero(PHI_T(1,1),np*(nxp-nyp));                                        ****
DO io=1,##C-1;IF (ko_p_typ(io).eq.1.and.ko_p_tvar(io).eq.1.and.ko_p_ini(io)
.eq.1)Phi_t(ko_p_kvar(io),io)=1.;
IF (ko_p_tvar(io).eq.1.and.ko_p_typ(io).ne.1)THEN;
write(*,*)''''***ERROR Cannot set Pert_type'''',ko_p_typ(io),
'''' on State variable:'''',EtaNam(ko_p_kvar(io));jo=1;ENDIF;
IF (ko_p_tvar(io).ne.1)THEN;write(*,*)''''***ERROR Transfer variable in sensy_to
_Eta block:'''';jo=1;ENDIF;ENDDO;
+SELF,IF=Flo_resp.  -----------------------------------------------------
IF(##C-1.NE.NP)THEN;WRITE(*,*)''''***ERROR all State Variable required in sensy
 to Eta block'''';jo=1;
! copie de Z dans Phit et de sigmaT dans son double
;####* === Initialiser : Z dans PhiT(1:n,1:n) ============;;
! *** A FAIRE ***
+SELF.
IF (jo.ne.0)write(loerr,*)'''' *** ERROR in ZINIT Sensy_to_Eta'''',char(7);'''
! -----------------------------------------------------------------------------
&';SENSY_TO_PHI<'=';WRITE(*,*);
WRITE(*,*)'' --------------- Informing on Sensitivity to Phi-variables ------'';
;WRITE(*,*)''  Var-name,             Perturbation-type,    column in Phit Mx'';
#L8#S1#L #S6#N; &''VAR:##,''=''##U6##S6;##1$VTF$;,##1$''
 &'',##$PERT:##;''='' ko_p_ini(nxp-nyp+##C)=1;ko_p_mod(nxp-nyp+##C)=-1;
 WRITE(*,''''(2A24,I3)'''')''''##1'''',''''##2'''',nxp-nyp+##U6##C;$PRT:##2;
##A1##S6;''
 &'',##$PERT:## AT''=''
 WRITE(*,''''(2A24,I3)'''')''''##1'''',''''##2'''',nxp-nyp+##U6##C;$PRT:##2;
##A1##S6;$AS'' &''$8$''=''##U6;##N;
 WRITE(*,*)'''' ------------------------------------------------------'''';
 &''''####K$8$'''' &''''####KVAR:####,'''' &''''####K,####$PRT:####;'''' jo=0;
Call veczero(PHI_T(1,nxp-nyp+1),np*nyp);
DO io=nxp-nyp+1,nxp-nyp2;IF (ko_p_tvar(io).eq.1)THEN;
write(*,*)''''***ERROR Eta variable in sensy_to_Phi block: ''''
,EtaNam(ko_p_kvar(io));jo=1;ENDIF;
IF (ko_p_tvar(io).ne.1.and.ko_p_typ(io).eq.1)THEN;
write(*,*)''''***ERROR Cannot set C.I. pert type for Transfer variable:''''
,PhiNam(ko_p_kvar(io));jo=1;ENDIF;ENDDO;
IF (jo.ne.0)write(loerr,*)''''*** ERROR in ZINIT Sensy_to_Phi'''',char(7);'''
! ----------------------------------------------------------------------------
;
```

```
  --------------- Informing on Sensitivity to variables ----------
   Var-name,              Perturbation-type,    column in Phit Mx
           T_ocean_eta                    init  1
           T_tropo_eta                    init  2
           Q_tropo_eta                    init  3
```

```
           T_WV_phi               step_resp  4
        T_Planck_phi              step_resp  5
          T_Evap_phi              step_resp  6
          T_sens_phi              step_resp  7
        T_FlxCMO_phi              step_resp  8
            xCO2_phi              step_resp  9
 -----------------------------------------------------
```

On décide que pour le calcul des vecterus de Floquet, on place Z dans $\Phi_t(t = 0)$ c'est-à-dire pas de chgt à sensy_to_var<>. On a déjà les rep-eff qui y sont demandées, de taille nyp, on double cette dimension pour y adjoindre la partie $\overline{\sigma(t)}$ derrière : nxp=[np]+2*nyp2, avec nyp=2*nyp2. dans ce cas de figure, le bloc sensy_to_var<> ne serait pas à modifier, on ajouterait seulement les initialisations, et on ajouterait aux dimensions nyp et nyp2. pSteer devra se limiter à nxp-nyp2.

Macros dodimetaphi :

```
! --------- al1 15/09/09 chgt #L -------
                          | ici pound sign pas codé par Latex latin1
&';SENSY_TO_VAR<'='#LS#S1#L#S6 &''##K;IF NXP.gt.0<''
 &'';VAR:##,##;''=''##U6##A1##S6;''
 &'';$S$;''='' &''''####K;$S$;'''' &''''####K;VAR:####,####;''''
+SELF,IF=-Flo_resp.   ---------------------------------------------------------
 &'''';parameter(nxp=0,''''='''';parameter(nxp=##U6##C,'''';'';'
+SELF,IF=Flo_resp.    ---------------------------------------------------------
 &'''';parameter(nxp=0,nyp=0''''=
'''';parameter(nxp=##U6##C+nyp,nyp=##C-np'''';'';'
+SELF.
&';SENSY_TO_ETA<'='#LS#S1#L#S6 &''##K;IF NXP.gt.0<''
 &'';VAR:##,##;''=''##U6##A1##S6;''
 &'';$S$;''='' &''''####K;$S$;'''' &''''####K;VAR:####,####;''''
+SELF,IF=-Flo_resp.   ---------------------------------------------------------
 &'''';parameter(nxp=####,''''='''' &''''''''########K;parameter(nxp=########,''
'''''';parameter(nxp=####1+##U6##C,'''';'';'
+SELF,IF=Flo_resp.    ---------------------------------------------------------
 &'''';parameter(nxp=0,''''='''';parameter(nxp=##U6##C+nyp,''''
;character*2 ozvf(np);'';'
+SELF.
&';SENSY_TO_PHI<'='#LS#S1#L#S6 &''##K;IF NXP.gt.0<''
 &'';VAR:##,##;''=''##U6##A1##S6;''
 &'';$S$;''='' &''''####K;$S$;'''' &''''####K;VAR:####,####;''''
+SELF,IF=-Flo_resp.   ---------------------------------------------------------
 &'''';parameter(nxp=####,nyp=0''''=''''
;parameter(nxp=####1+##U6##C,nyp=##C'''';'';'
+SELF,IF=Flo_resp.    ---------------------------------------------------------
 &'''';parameter(nxp=####,nyp=0,nyp2=0''''=''''
;parameter(nxp=####1,nyp=2*##U6##C,nyp2=##C'''';'';'
+SELF.
! -----------------------------------------------------------------------------
Il vaut mieux calculer eta et ff indépendamment *** cf ci-dessus
et en accord :  (il n'y a que nxp-nyp2 variables sensibles) mais nxp colonnes
(mais les modifs sont OK pour sensy-to-var quand même)

! extension pour Phi_t et Psi_t (propagateurs TEF)
```

```
   if nxp.gt.0
   < OPEN(loPhit,FILE='sens'//osuffix,STATUS='UNKNOWN');
      write(loPhit,1011)' Sensy_to:',
+SELF,IF=-Flo_resp.
       (EtaNam(np*(ko_p_tvar(jo)-1)+ko_p_kvar(jo))
                            ,(charit2(io+2+np*(jo-1)),io=1,np-1),jo=1,nxp)
+SELF,IF=Flo_resp.
       (ozvf(jo)         ,(charit2(io+2+np*(jo-1)),io=1,np-1),jo=1,nxp-nyp)
                             ,('$'//EtaNam(np*(ko_p_tvar(jo)-1)+ko_p_kvar(jo))
                 ,(charit2(io+2+np*(jo-1)),io=1,np-1),jo=nxp-nyp+1,nxp-nyp2)
                              ,(EtaNam(np*(ko_p_tvar(jo-nyp2)-1)+ko_p_kvar(jo-nyp2))
                       ,(charit2(io+2+np*(jo-1)),io=1,np-1),jo=nxp-nyp2+1,nxp)
+SELF.
      ;write(loPhit,1011)' time \\  of:',((EtaNam(io),io=1,np),jo=1,nxp)
     ;OPEN(loPsit,FILE='sigma'//osuffix,STATUS='UNKNOWN');
! ---- Psi_t out -------------
      write(loPsit,1011)' Sensy_to:',
+SELF,IF=-Flo_resp.
       (EtaNam(np*(ko_p_tvar(jo)-1)+ko_p_kvar(jo))
                              ,(charit2(io+2+mp*(jo-1)),io=1,mp-1),jo=1,nxp)
+SELF,IF=Flo_resp.
       (ozvf(jo)           ,(charit2(io+2+mp*(jo-1)),io=1,mp-1),jo=1,nxp-nyp)
                              ,('$'//EtaNam(np*(ko_p_tvar(jo)-1)+ko_p_kvar(jo))
                   ,(charit2(io+2+mp*(jo-1)),io=1,mp-1),jo=nxp-nyp+1,nxp-nyp2)
                        ,(EtaNam(np*(ko_p_tvar(jo-nyp2)-1)+ko_p_kvar(jo-nyp2))
                   ,(charit2(io+2+mp*(jo-nyp-1)),io=1,mp-1),jo=nxp-nyp2+1,nxp)
+SELF.
      ;write(loPsit,1011)' time \\  of:',((PhiNam(io),io=1,mp),jo=1,nxp);
    ;if mobs.gt.0
     < OPEN(loPsot,FILE='sigmo'//osuffix,STATUS='UNKNOWN');
! ---- parameter sensy ----------------
      write(loPsot,1011)' Sensy_to:',
+SELF,IF=-Flo_resp.
     (EtaNam(np*(ko_p_tvar(jo)-1)+ko_p_kvar(jo))
                           ,(charit2(io+2+mobs*(jo-1)),io=1,mobs-1),jo=1,nxp)
+SELF,IF=Flo_resp.
     (ozvf(jo)           ,(charit2(io+2+mobs*(jo-1)),io=1,mobs-1),jo=1,nxp-nyp)
                               ,('$'//EtaNam(np*(ko_p_tvar(jo)-1)+ko_p_kvar(jo))
                ,(charit2(io+2+mobs*(jo-1)),io=1,mobs-1),jo=nxp-nyp+1,nxp-nyp2)
                        ,(EtaNam(np*(ko_p_tvar(jo-nyp2)-1)+ko_p_kvar(jo-nyp2))
                ,(charit2(io+2+mobs*(jo-nyp-1)),io=1,mobs-1),jo=nxp-nyp2+1,nxp)
+SELF.
      ;write(loPsot,1011)' time \\  of:',((PhiNam(mp+io),io=1,mobs),jo=1,nxp)
+SELF,IF=Flo_resp.
! --- Floquet response components ----------
 OPEN(loFloS,FILE='Flo_sig'//osuffix,STATUS='UNKNOWN');
      write(loFloS,1011)' Sensy_to:',
       ( (ozvf(ko)//'$'//EtaNam(np*(ko_p_tvar(jo)-1)+ko_p_kvar(jo))
                   ,(charit2(io+2+np*(ko-1+nfp*(jo-nxp+nyp-1))),io=1,np-1)
                                                ,ko=1,nfp),jo=nxp-nyp+1,nxp-nyp2);
      write(loFloS,1011)' time \\  of:',((EtaNam(io),io=1,np),jo=1,nfp);
;
```

```
 OPEN(loFloT,FILE='Flo_rho'//osuffix,STATUS='UNKNOWN');
      write(loFloT,1011)' Sensy_to:',
       ( (ozvf(ko)//'$'//EtaNam(np*(ko_p_tvar(jo)-1)+ko_p_kvar(jo))
                    ,(charit2(io+2+mp*(ko-1+nfp*(jo-nxp+nyp-1))),io=1,mp-1)
                                           ,ko=1,nfp),jo=nxp-nyp+1,nxp-nyp2);
      write(loFloT,1011)' time \\  of:',((PhiNam(io),io=1,mp),jo=1,nfp);
! --- partie Observations ---
if mobs.gt.0
< OPEN(loFloM,FILE='Flo_robs'//osuffix,STATUS='UNKNOWN');
      write(loFloM,1011)' Sensy_to:',
       ( (ozvf(ko)//'$'//EtaNam(np*(ko_p_tvar(jo)-1)+ko_p_kvar(jo))
                    ,(charit2(io+2+mobs*(ko-1+nfp*(jo-nxp+nyp-1))),io=1,mobs-1)
                                           ,ko=1,nfp),jo=nxp-nyp+1,nxp-nyp2);
      write(loFloM,1011)' time \\  of:',((PhiNam(mp+io),io=1,mobs),jo=1,nxpx);
>;
SELF.
     >;
   >;
```

On a en plus les tableaux de pointeurs et de calcul à la fois pour Floquet et 2. Dans le cas où on ne s'intéresse qu'à une partie des Floquet, Floquet2 reste OK, mais il y aura un travail spécifique dans un Floquet2 pour extraire l'information tronquée des pointeurs — si c'est les premiers, facile ...

question subs : Que se passe-t-il avec des variables constantes de la liste des états ? Peut-on dans ce cas restreindre la dim pour le calcul complet ?

Pour les notations, on reprend les noms de variable avec un $ en préfixe ($T_WV_Eta), et on crée un tableau character*2 ozvf(np) pour les Floquets, qui disent si r# (reel) ou R# I# (cmplx).

On aurait alors pour un calcul de gains :

```
 Sensy_to_Eta
 <
           var: T_ocean_eta   , pert: init;
           var: T_tropo_eta   , pert: init;
           var: T_strato_eta  , pert: init;
           var: Q_tropo_eta   , pert: init;
           var: CO2_conc_eta  , pert: init;
 >;
! Rep-eff rétroaction
 Sensy_to_Phi
 <         var: T_WV_phi      , pert: step_eff;
 >;
! moderateur sensib a xCO2 :
!***  Phi_t(5,5)=3.30d-04; "XXX"
       step_psi(6)=3.30d-04;
```

en gardant sensy-to-var pour le standard (les macros tueront celles qu'il faut). Rq : faudra dans ce cas tuer les sources ?

le calcul des composantes On a montré (actuellement dans WVsimple) une décomposition de l'intégrale donnant directement la forme efficace :

$$\overline{\sigma} = \int_t^{t+T} \Phi(t+T,\tau) \mid b(\tau) > s(\tau) d\tau \tag{22}$$

qui fournit finalement

$$\overline{\sigma}(t) = \Phi(t,0)\sigma^F(T) + \int_0^t \Psi^F(t)e^{\Lambda(t-\tau)}[I - e^{\Lambda T}]\Psi^{-F}(\tau) \mid b(\tau) > s(\tau)d\tau \qquad (23)$$

Si bien que l'algorithme complet consiste dans les calculs successifs suivants

1. avec la première détermination de 0 à T **sequence Floquet** activées :
   - le propagateur complet $\Phi(t,0)$,
   - les sensibilités rep-eff pour chaque réponse à calculer ($\rightarrow \mid b_i >$) dont on conserve $\sigma_i(T)$;
   - avec $\Phi(T,0)$ le calcul de Z et Λ (init Floquet standard) ;
2. une période pour avancer **sequences Flo_vec et Flo_reps** activées :
   - les Floquet $Z \rightarrow \Psi_r^F$ et les rep-eff $\sigma(t)$ rangés dans le tableau Phi_t standard $(np \times np + nyp2)$;
   - en même temps, les $\sigma(T) \rightarrow \sigma^F(t)$,
   - les $\sigma(T) \rightarrow \sigma^F(t)$ en extension de Phi_t$(np \times nyp2)$;
3. et en fin d'incrément :
   - calcul effectif des Ψ^F de leur complément complexe, et alors de Ψ^{-F},
   - calcul des composantes de l'effet-en-retour asymptotique $\overline{\sigma}(t)$ puis de

$$\overline{\sigma}_k(t) = \mid \psi_k^F(t) >< \psi_k^{-F}(t) \mid [\ \sigma^F(t) + (1 - e^{\lambda_k T})\sigma(t,0)\ ]\ ; \qquad (24)$$

     par paires conjuguées ;
   - les transferts standards avec en plus les composantes de $\overline{\varrho_k}$;

On a encore montré l'efficacité de la réélification de Φ^F qui remplace (24) par

$$\overline{\sigma}_k(t) \quad = \mid u,v(t) >< x,y(t) \mid_k \sigma^F(t) + \mid u,v >_k [L(t)]_k < x,y \mid_k \sigma(t,0) \qquad (25)$$
$$\text{avec } [L] = \begin{bmatrix} 1 - e^{\lambda T} & e^{\mu T} \\ -e^{\mu T} & 1 - e^{\lambda T} \end{bmatrix}$$

pour chaque paire et ses exposants, et son correspondant diagonal pur pour chaque V.F.réel, d'où l'algorithme suivant.

```
  parameter (nxpx=nfp*nyp2,nxpz=nxp+nxpx);
  real oxy_inv(np,np),owmxn(np,np),owmxmy(mobs,nxpx);
  real owrow(nyp),oSig_b(np,nxpx),oRep_b(mp,nxpx);
autres modifs :
      real b_c_v_adj(mp),d_c_w_mu(mp),Ctbuf(mp+mobs,np+lp)
!                                     **** revoir les buffers et en equiv.

!# #################
+SELF, IF=FLO_RESP.
!##################
!  Epreuve d'algorithme a rentrer dans le code standard
! -------------------------------------------------------
! Calcul de  sigma_b = sigma_F + (I-Lambda T) sigma de k
! -----------------------------------------------------
! Tableau Phi_t == PhiF   , Phi 1  , Phi 2   np x
!                  np       nyp2     nyp2
!                FloquetR, sigma  ,  sigma_F
! -----------------------------------------------------
! Sorties : pour chaque rep_eff, ses nfp composantes
! -----------------------------------------------------
! cols PhiF =           |u> ou |u,v>
! a1a = inv PhiF lignes <x| ou <x/y|
! -----------------------------------------------------
 call omatcopy(n,n,Phi_t(1,1),np,owmxn,np);
 call diagmat(one,oxy_inv,n,n);
 call Sgesv(np,np,owmxn,np,iPiv,oxy_inv,np,infores);
;
  Do ((i=1,np),j=1,nxpx)< oSig_b(i,j)=0.; >;
  Do io=1,2*nfp,2 "nfp fixe le nbre de composantes d'interet (parameter)"
  <
   ko=ko_colF(io); "ptr 1st Cx col, or Rl col"
   lo=(io+1)/2;    " ptr composante"

!    Calcul du facteur correctif
!    --------------------------
   wox = Flo_sigma(ko);
   aor = 1.-real(wox); bor = imag(wox);
;
!    Correction du cas complexe
!    -------------------------
    if ko_colF(io+1).eq.1                    "cas complexe"
    <
! definition des |u> et |v> (pour clarte)
     call omatcopy(n,1,Phi_t(1,ko),np,orvec,np);
     call omatcopy(n,1,Phi_t(1,ko+1),np,oivec,np);
! premiere partie
! ................
! Ia/  <x|[Phi1,Phi2] -> <z1,z2|
      call omatmlt(1,np,nyp,oxy_inv(ko),np,Phi_t(1,np+1),np,owrow,1);
! Ib/  |u><z1| + |aor u -bor v><z2| -> Sig_b
      Do ((i=1,np),j=1,nyp)
      < oSig_b(i,lo+(j-1)*nfp) = oSig_b(i,lo+(j-1)*nfp)
```

```
                            + orvec(i)*owrow(nyp2+j)
                            + (aor*orvec(i) - bor*oivec(i))*owrow(j);
      >;
! deuxieme partie
! ................
! IIa/ <y|[Phi1,Phi2]
      call omatmlt(1,np,nyp,oxy_inv(ko+1,1),np,Phi_t(1,np+1),np,owrow,1);
! IIb/ |v><z1|+ |bor u +aor v><z2| -> +Sig_b
      Do ((i=1,np),j=1,nyp)
      < oSig_b(i,lo+(j-1)*nfp) = oSig_b(i,lo+(j-1)*nfp)
                                   + oivec(i)*owrow(nyp2+j)
                             + (bor*orvec(i) + aor*oivec(i))*owrow(j);
      >;
   ;
    >elseif ko_colF(io+1).eq.-1                   "cas reel"
    <
!    ---------------------------
!     Correction du cas reel ----
!    ---------------------------
! definition de  |u>        (pour clarte)
     call omatcopy(n,1,Phi_t(1,ko),np,orvec,np);
! a/   <x|[Phi1,Phi2] -> <z1,z2|
      call omatmlt(1,np,nyp,oxy_inv(ko,1),np,Phi_t(1,np+1),np,owrow,1);
! b/   |u><z1 + aor z2|            -> Sig_b
      Do ((i=1,np),j=1,nyp)
      < Do k=1,n
        < oSig_b(i,lo+(j-1)*nfp) = oSig_b(i,lo+(j-1)*nfp)
                      + (orvec(i)*oxy_inv(ko,k))*(Phi_t(k,np+nyp2+j)
                                                 + aor*Phi_t(k,np+j));
        >;
!-                       +  orvec(i)*"("owrow(nyp2+j) "+ aor*owrow(j))";
      >;
    >elseif ko_colF(io+1).eq.-2         "cas mult reel negatif"
    < "a traiter car on a bien un V-F Cplx !"
! ..........................................
    >;"EndIf Cx/Rl"
;
  >;"end Do"
```

Alors, le mieux est de modifier la partie "calcul des sensibilités" qui détermine Ψ d'après Φ, car on est alors à jour des matrices (η, φ). Il suffit d'une extension à nxp du **parameter nxpz=nxp+nfp*nyp2**. Il faut modifier :

```
! Calcul des resp_eff du pas precedent (Ct et D(t))
! ----------------------------------------------------
!             (I-D) rho_b = Ct oSig_b
! ----------------------------------------------------
  call omatmlt(m,n,nxpx,Ct,mp+mobs,oSig_b(1,1),np,oRho_b(1,1),mp+mobs);
  call omatmlt(m,m,nxpx,un_moins_d,mp,oRho_b(1,1),mp+mobs,oRho_b(1,1),mp+mobs);
;
! output sig et rho
! -----------------
if Zout
```

```
< write(loFloS,1000) time,oSig_b; write(loFloR,1000) time,oRho_b;
>;

! --- partie Obs : rho_mu =  c2 sig_b + d_fmu rho_b
  if mobs.gt.0
  < call omatmlt(mobs,n,nxpx,Ct(mp+1,1),mp+mobs,oSig_b(1,1),np  "c2 sig_b"
                                            ,oRho_b(mp+1,1),mp+mobs);
    call omatmlt(mobs,m,nxpx,D(mp+1,1),mp+mobs,oRho_b(1,1),mp+mobs
                                ,Ctbuf,mobs);
    call omatadd(mobs,nxpx,Ctbuf,mobs,oRho_b(mp+1,1),mp+mobs
                                               ,oRho_b(mp+1,1),mp+mobs);
if Zout
< write(loFloR,1000) time,((oRho_b(mp+i,j),i=1,mobs),j=1,nxpx);
>;
  >;
```

Index

Table des matières