Manuel pratique de Mini-ker

Al1, Janvier 2007*

Versions 1.02.xx

Ce manuel est une introduction à l'utilisation pratique de l'environnement Mini_ker dans sa version 1.02. Le lecteur est renvoyé aux différents documents concernant le TEF, l'adjoint et les sensibilités sur la page **ZOOM** *en note de bas de page 2. Signalons également l'existence d'un cahier de programmation — sur demande.*

La version 1.02 comporte les facilités suivantes :

- passage immédiat en double précision à la demande ;
- calcul des sensibilités en avant des variables à une liste de paramètres[1] (**free_parameter**) ; plusieurs exemple de fit par cette méthode ont été effectuées (cf rapport Céline de Mai 2007, et les exos Evex et ClimSI).
- calcul des sensibilités de la fonction de coût à la même liste de paramètres (**free_parameter, Zback**) ;
- un vecteur des "observables" étendant le vecteur des transferts (**set_probe**<) ;
- le calcul de sensibilité en avant des variables à plusieurs types de perturbation (**Sensy_to_var**<). Ceci permet notamment de calculer le propagateur et les exposants de Lyapunov le long de la trajectoire (cf exo de stepH), mais aussi le gain de rétroaction non linéaire sous la forme réponse : $\frac{1}{1-g}(t,0)$;
- calcul de la sensibilité de la matrice d'avance d'état **dPi_aspha** au **fwd** paramètre Ce calcul (symbolique) proposé par stb permet ensuite de calculer la sensibilité au paramètre choisi, soit des valeurs propres, soit des valeurs singulières, de diverses matrices dérivées des Jacobiennes ;
- utilisation d'un filtre de Kalman sur une base de données (Pat) ; le filtrage a été amélioré (cf cahiers Propagateurs) pour assurer le mieux possible la positivité stricte de l'avance de la var-covar ;
- recherche d'optimum paramétriques ou de loi de commande avec Minuit[2].
- calcul des fonctions SLT (gain, réponses) en Borel par balayage en dt ou par la méthode des éléments propres et inversion en variable temporelle (cf routines Bsweep et Boreleig).
- dimensionnement automatique (ex DimEtaPhi générée) ;

Préannonce de la 2.00

Dans l'optique d'un développement avec composante industrielle, on n'envisage pas de version 1.03, mais directement celle d'une maquette (2.00) comportant un seul emboîtement de familles - à la ZOOM.

Ceci devrait être l'objet de discussions dans les mois à venir, avec ou sans l'accord de l'ANR á laquelle nous avons soumis le projet **Dyams**.

*avec des contributions de pat, stepH , stb, seb, jlj, jyg et ribs

[1] On devrait alors dans la foulée proposer en standard la méthode de Newton pour offrir une alternative au fit de paramètres avec Minuit et ainsi réhabiliter LLSQ — Least Linear Square fit par méthode de Householder, routine du *Paris Sud Informatique* Université de Paris-Sud.

[2] ©CERN

Principes et structure du Mini_ker

Un modèle formulé sous TEF sépare deux types d'équations : a) des équations d'évolution référant aux modèles de **cellules** et : b) des modèles dits de **transferts** exprimant des contraintes statiques. De façon générale, ces objets associent à chaque variable une équation (un modèle) et ces objets sont rangés en familles formant une structure d'arbre[1]. Le traitement complet des modèles sous TEF renvoie à l'environnement logiciel **ZOOM**[2].

Mini_ker constitue une entrée plus facile à la modélisation sous TEF en simplifiant la structure du modèle : une seule cellule, un seul transfert, une seule famille. Alors que ZOOM peut traiter de l'ordre de 100 000 équations, Mini_ker n'en peut traiter pratiquement que quelques centaines.

Mini_ker effectue automatiquement les calculs de sensibilité aux conditions initiales ou à un paramètre du modèle (en avant). Le modèle adjoint est automatiquement généré, qui permet (en arrière) de calculer la sensibilité d'une fonctionnelle de coût aux variables du modèle et à ses paramètres.

Toutes les dérivées partielles nécessaires à la résolution du système (et des sensibilités et de l'adjoint) par le TEF sont effectuées symboliquement à la précompilationi. L'utilisateur a donc terminé la programmation de son modèle dès qu'il a entré les fonctions décrivant le modèle et donné la valeur des paramètres ainsi que les conditions initiales. La programmation pratique est décrite dans ce manuel en première section.

Le langage développé pour automatiser le développement du TEF à partir des fonctions du modèle utilise **Mortran**, un pré-compilateur Fortran développé dans les années 1970 (cf manuel Mortran). La première annexe en explique le principe d'utilisation pour le Mini_ker

En section 5, on indique comment se servir des différents outils d'analyse des systèmes dynamiques traités, qui exploitent la formulation par le TEF du modèle. L'utilisation de Minuit en complèment de l'adjoint est présentée en section 6, et l'utilisation de données est étendue avec le filtre de Kalman (7) et l'option **ZObs**.

1 L'entrée d'un modèle dans Mini_ker

Deux niveaux de langage étant disponibles, nous décrivons d'abord le langage le plus proche de la résolution car il utilise une correspondance immédiate entre les vecteurs η, φ du TEF et les tableaux Fortran correspondants **eta(.)** et **ff(.)**. Le deuxième niveau est alors introduit, qui permet une écriture plus proche du modèle symbolique, et donne également accès aux facilités pour mettre en œuvre un maillage uni-dimensionnel.

1.1 Principe d'organisation du code

Le programme principal de Mini_ker est subdivisé en morceaux, que l'on apelle séquences. L'utilisateur peut se contenter d'un accès à un nombre réduit de séquences dans lesquelles il code, en pseudo-Fortran, les instructions de dimentions du modèle et son expression mathématique. Lors de l'assemblage, ces instructions sont traduites par Mortran en Fortran à l'aide de macros qui remplacent, dans le corps des calculs, des instructions symboliques par du code piloté par la description du modèle.

Les séquences principales sont rangées dans un fichier. Nous utilisons ici implicitement le langage de **cmz**, logiciel de gestion de sources et de fabrication d'exécutables du CERN[3].

[1] cf http :/www.lmd.jussieu.fr/documents.dir/tef-GB-partA5.ps.gz
[2] cf http :/www.lmd.jussieu.fr/zoom
[3] une version sans cmz, portable sur Windaube et Mac OSX a été développée par Pat, se référer à la version anglaise du manuel.

Ainsi, une séquence est un élément de code formant une **+KEEP** de cmz, équivalent à un fichier file.h du préprocesseur Fortran. Les séquences à programmer sont **$dimetaphi** pour entrer les dimentions des tableaux eta(.) et ff(.), et **$Zinit** pour programmer le modèle. Deux autres séquences utiles (**$Zsteer, $ZStep**) sont introduites à la fin de la boucle d'avance temporelle du programme principal et utilisées pour effectuer des test et calculs numériques suuplémentaires, gérer le niveau d'impression, interrompre la simulation etc.

1.2 Description basique du modèle

On démarre avec une cmz file du nom de son exercice (exemple donné : **predator** - en général, on part d'un des nombreux exemples fournis que l'on va modifier). Sous CMZ, on va dans le patch **ZinProc**[1]. Un modèle formulé sous le TEF conduit au système matriciel :

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

Le modèle prédateur-proie de Lotka-Volterra peut s'écrire sous la forme suivante :

$$\begin{cases} \partial_t \eta_{prey}(t) = & a\eta_{prey} - a\varphi_{meet} \\ \partial_t \eta_{pred}(t) = & -c\eta_{pred} + c\varphi_{meet} \end{cases}$$

$$\varphi_{meet} = \eta_{prey}\eta_{pred} \tag{2}$$

dans laquelle on a deux équations d'état, c'est-à-dire une équation d'évolution pour chaque groupe proie et prédateur, et un seul transfert représentant les rencontres entre individus de groupes adverses.

séquence dimetaphi

La séquence où on doit rentrer les dimensions des vecteurs eta(.) et ff(.) se trouve dans **$dimetaphi**, il s'agit respectivement des **parameter np** et **mp**.

```
+KEEP,dimetaphi.
! +++++++++++++++++++++++++++++++++++++++++++++++++++++++
!       nbre de variables d'etat       : np            +
!       nbre de variables de transfert : mp            +
!       nbre de parametres libres      : lp            +
! +++++++++++++++++++++++++++++++++++++++++++++++++++++++
      parameter (np=2,mp=1);
      parameter (lp=0);
! ------------------------------------------------------
+KEND.
```

Pour cet exemple, on a deux états et un seul transfert (et pas de paramètre libre).

Pour que Mini_ker retienne ce calcul manuel, il faut à présent lever dans sa selseq.kumac[2] le drapeau dimetaphi (**sel dimetaphi**). Sinon, il est inutile de remplir cette séquence, le dimensionnement sera automatiquement effectué par Mini_ker.

séquence ZINIT

Le modèle se programme dans **$Zinit**. On rentre d'abord les paramètres du modèle :

```
!%%%%%%%%%%%%%%%%%%%%%%
! Parameters                            "! = carte commentaires        "
!%%%%%%%%%%%%%%%%%%%%%%


     dt=.01;                            "pas de temps initial          "
```

[1]**cd zinproc** puis **ls** pour lister le contenu du patch.

[2]fournie avec les fichiers Mini_ker, cf ftp anonyme pub/alain.

```
     nstep=10 000;                          "nbre d'itérations              "
     modzprint = 1000;                      "fréquence des printouts        "
;
     time=0.;                               "initialisation du temps modèle "
;
! parametres du modele                     "le langage est du pseudo Fortran"
     apar = 1.5;                            "avec des fins d'intructions    "
     cpar = 0.7;                            "signalées par ';'              "

print*,'**************************************';
print*,'Lotka-Volterra model with parameters as:';
z_pr: apar,bpar;
print*,'**************************************';
```

Une carte débutant par le signe "!" est un commentaire, les instructions Mortran sont de format libre et terminées par un point-virgule. On a défini le pas de temps initial dt, qui s'ajoutera à la variable **time** du temps de la simulation à la fin du premier pas de temps[1]. Les paramètres du modèles sont aussi donnés.

On remarque l'instruction d'impression **z_pr** dont la macro Mortran est en tête de la séquence ZINIT dans la cmz file de chaque exemple[2]. Les instructions ci-dessus produisent l'impression :

```
 **************************************
 Lotka-Volterra model with parameters as:
    apar,cpar :
      1.5  0.699999988
 **************************************
```

On définit à présent le modèle, c'est-à-dire ici une seule composante ff(1), et deux d'états. On programme directement les expressions à droite du signe égal des équations (2), composante par composante, en utilisant la macro **Fun_set** (*function set*) :

```
!%%%%%%%%%%%%%%%%%%%%%%
! Transfer definition
!%%%%%%%%%%%%%%%%%%%%%%
! ff(1) : rencontres
    fun_set Phi_tef(1) = eta(1)*eta(2);                  "modèle de transfert"
!%%%%%%%%%%%%%%%%%%%%%%
! Cell definition
!%%%%%%%%%%%%%%%%%%%%%%
! eta(1) : pray
! eta(2) : predator                                     "modèles de cellule"
    fun_set  deta_tef(1) =   apar*eta(1) - apar*ff(1);
    fun_set  deta_tef(2) = - cpar*eta(2) + cpar*ff(1);

!     Etat initial
!     ------------
     eta(1) = 1.;
     eta(2) = 1.;
;
    OPEN(50,FILE='title.tex',STATUS='UNKNOWN');   "utilisé par SLTC,SLTCIRC"
    write(50,5000) apar,cpar;                              "et Minuik    "
5000;format('Lotka-Volterra par:',2F4.1);
```

Les états sont aussi initialisés, et on provoque l'écriture du titre choisi dans le fichier **title.tex** qui est obligatoire, car il sert à transmettre de l'information aux sorties graphiques.

Les noms de fonctions **dEta_tef** et **Phi_tef**[3] sont réservés, leur numérotation est en correspondance avec les vecteurs eta(.) et ff(.). La simulation sans calculs de sensibilité ou

[1] on fera varier ce pas de temps dans **ZSTEER**

[2] ceci pour suggérer à l'utilisateur qu'il est très simple de se mettre aussi a faire ses propres macros Mortran.

[3] la convention majuscules-minuscules indistinctes est celle du Fortran.

de l'adjoint est prête[1]. En ce qui concerne le symbole des noms de variables introduites par l'utilisateur, et pour éviter les symboles communs avec ceux du reste du code, on a adopté la convention d'interdire le caractère "o" dans les deux premiers caractères des noms de variables Fortran.

Remarque sur l'initialisation des transferts : *dans le cas où le modèle comporte une matrice D (c'est-à-dire qu'il y a une relation implicite entre des composantes de transferts) il peut être utile voire nécessaire d'initialiser quelques transferts. Cet aspect ne regarde que la toute première itération de la résolution du système des transferts. On a intéret à faciliter cette première passe en initialisant les transferts-paramètres ainsi que les variables de transferts, qui prendront sinon la valeur nulle comme première estimation. Cela peut donc conduire à problème numérique si l'inverse d'une variable apparait dans les formules, ou plus généralement rend une fonction singulière.*

On aura intérêt en cas de problème à lever le drapeau **sel debug** *dans la selseq.kumac pour : a) recupérer en sortie une trace de la D-Loop, et b) activer un calcul d'estimation de conditionnement de la matrice D avant inversion (par LaPack). On pourra également visualiser la variation itérative des transferts dans le fichier* **tr_in_dloop.data**. *L'option* **debug** *est donc très bavarde, ne pas oublier de réduire* **nstep**.

1.3 Lancement de la simulation et sorties

On lance la précompilation, compilation et fabrication de l'exécutable en tapant **mod** dans la fenêtre CMZ. L'éxécution de cette macro va assembler les parties du code fragmenté en séquences, pour former un programme principal en langage Mortan, que l'on trouve dans, avec l'exemple predator : predator/predator/cfs/predator_o.tmp. On retrouve dans ce fichier ses séquences dimetaphi et zinit, zsteer en fin de la boucle de calcul, immergées dans le programme **principal**. L'assemblage est guidé par l'appel de mod à une macro cmz : **selseq.kumac**, située dans le répertoire de la cmz file. C'est dans cette kumac que l'on donne le path d'accès au code complet de Mini_ker, qui impose les séquences de l'utilisateur, et qui aiguille les options diverses décrites plus loin.

On a par ailleurs le résultat du passage de Mortran dans le fichier : predator/predator/cfs/principal.f . Après **mod** sous cmz, un exécutable doit être présent sous le répertoire **predator/predator** par exemple. On lance cet éxécutable

predator.exe >res.lis

On obtient en sortie des fichiers .data et le listing (res.lis). Chaque fichier **.data** comporte une première colonne donnant la variable **time** [2]. Les colonnes suivantes donnent les valeurs de **eta(.)** dans res.data, **dEta(.)** dans dres.data, la variation de eta(.) à chaque pas de temps, **ff(.)** dans tr.data. On a enfin sur sélection de l'option **sel doteta** dans la selseq.kumac un fichier doteta.data donnant la vitesse $\partial_t \eta$. D'autres fichiers **.data** seront décrits plus loin.

De manière analogue à l'option modzprint / Zprint, l'écriture des fichiers .data peut être commandée par le couple modzout / Zout (cf **ZSteer**).

[1] *Il est également possible de programmer soi-même toutes les matrices Jacobiennes et vecteurs du TEF en entrant dans son zinproc les séquences D_Tr, D_Mx, A_Mx, B_Mx, Ct_Mx, Omega_vec et Gamma_vec à la manière du Mini_ker originel de stb. On attend d'avoir des exemples pour décrire ce mode de programmation un peu près du filet.*

[2] **dres.data** comporte une deuxième colonne donnant **dtime** qui a pu varier en cours de simulation.

1.4 Description symbolique du modèle

Une couche de langage supplémentaire permet une description plus symbolique d'un modèle[1], mais qui résultera exactement dans les mêmes vecteurs et fonctions de base précédentes (et un programme Fortran identique). L'exemple déjà donné peut en effet s'écrire de façon équivalente :

```
!%%%%%%%%%%%%%%%%%%%%%%
! Transfer definition
!%%%%%%%%%%%%%%%%%%%%%%
! rencontres
set_Phi
< var: ff_interact, fun: f_interact = eta_pray*eta_pred;
>;

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

set_eta
< var: eta_pray, fun: deta_pray =   apar*eta_pray - apar*ff_interact;
  var: eta_pred, fun: deta_pred = - cpar*eta_pred + cpar*ff_interact;
>;
```

Ces instructions génèrent en Fortran les instructions d'impression de la correspondance entre les noms symboliques et les vecteurs et fonctions de base. Dans cet exemple, on aura ainsi en sortie :

```
  ---------------- Informing on Phi definition -----------------
    Var-name,           Function-name,       index in ff vector
           ff_interact              f_interact  1
  ----------------------------------------------------

  --------------- Informing on Eta definition ------------------
   Var-name,           Function-name,       index in eta vector
              eta_pray              deta_pray  1
              eta_pred              deta_pred  2
  ----------------------------------------------------
```

Enfin une dernière simplification d'écriture est possible :

```
!%%%%%%%%%%%%%%%%%%%%%%
! Transfer definition
!%%%%%%%%%%%%%%%%%%%%%%
! rencontres
set_Phi
< eqn: ff_interact = eta_pray*eta_pred;
>;

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

set_eta
< eqn: eta_pray =   apar*eta_pray - apar*ff_interact;
  eqn: eta_pred = - cpar*eta_pred + cpar*ff_interact;
>;
```

où les fonctions associées auront un nom implicite **$ff_interact** etc.

[1] sur la suggestion de jlj.

1.5 L'Options Grid1D

Des macros spécifiques ont été définies pour rentrer simplement un modèle maillé, limité à une seule dimension. En plus de la définition générique des modèles attachés aux mailles, il est nécessaire en général de définir des conditions aux limites, ou des sources ou forçages de part et d'autre du maillage.

Nous présentons l'exemple d'une chaîne de maselottes[1] reliées entre elles par des ressorts et amortisseurs, limitée à gauche par un mur et libre à droite. La formulation TEF du problème s'écrit dans l'espace de phase (position, velocité) du nœud k :

$$\left\{ \begin{array}{c} \partial_t \eta_k^{pos} = \eta_k^{vel} \\ \partial_t \eta_k^{vel} = (\varphi_k^{spr} - \varphi_{k+1}^{spr} \quad + \varphi_k^{dmp} - \varphi_{k+1}^{dmp}) \, / m_k \end{array} \right.$$

$$\left\{ \begin{array}{l} \varphi_k^{spr} = -k_k(\eta_k^{pos} - \eta_{k-1}^{pos}) \\ \varphi_k^{spr} = -d_k(\eta_k^{vel} - \eta_{k-1}^{vel}) \end{array} \right. \tag{3}$$

où m_k est la masse du nœud k, r_k et d_k la rigidité des ressorts et les coefficients d'amortissement.

séquence ZINIT

La convention de numérotage des mailles (pour cet exemple) est décrite en commentaire du code suivant :

```
!%%%%%%%%%%%%%%%%%%%%%%
! Parameters
!%%%%%%%%%%%%%%%%%%%%%%
dimension rk(n_node),rd(n_node),rmassm1(n_node);
data rk/n_node*1./;
data rd/n_node*0.1/;
data rmassm1/n_node*1./;
     dt=.01;       time=0.;
     nstep=5 000;  modzprint = 1000;
!%%%%%%%%%%%%%%%%%%%%%%%%%%================================================
! Maillage convention inode
!%%%%%%%%%%%%%%%%%%%%%%%%%%                                 Open ended
!(2 Lo      Phi    Eta                         (n_node)
! Eta)  \|       .-----.       .-----.          .-----.        /
! wall  \|-\/\/\-|     |-\/\/\-|     |  . . .  -|     |-\/\/\- |dummy
!  pos  \|--***--|  1  |--***--|  2  |  . . .  -|  n  |--***-- |Phis
! speed \|   1   |_____|   2   |_____|      n   |_____|  n+1   \(2 Hi Phi)
! Dwn                                                           Up
!%%%%%%%%%%%%%%%%%%%%%%====================================================
! Cell definition
!%%%%%%%%%%%%%%%%%%%%%%

! LO cells (wall)
! ----------------
set_dwn_eta
< var: eta_pos_wall,  fun: deta_pos_wall  = 0.;
  var: eta_speed_wall, fun: deta_speed_wall= 0.;
>;
! node cells
! ----------
set_[3]node_Eta
< var: eta_move(inode),  fun: deta_move(inode) = eta_speed(inode);
  var: eta_speed(inode),
  fun: deta_speed(inode) = rmassm1(inode)
                             *( - ff_spring(inode+1) + ff_spring(inode)
                               - ff_dump(inode+1)  + ff_dump(inode)    );
```

[1] l'exemple est fournit dans **masselottes**.

```
>;
!---------------
! Initialisation
!---------------
eta_pos_wall = 0; eta_speed_wall = 0.;
<inode=1,n_node; eta_move(inode)=0.01; eta_speed(inode)=0.0;>;
;
Z_Pr: eta;
!%%%%%%%%%%%%%%%%%%%%%%
! Transfer definition
!%%%%%%%%%%%%%%%%%%%%%%
! Hi limiting Transfers : dummy ones
! ---------------------------------
set_Up_Phi
< var:ff_dummy_1, fun: f_dummy_1=0.;
  var:ff_dummy_2, fun: f_dummy_2=0.;
>;
! node transfers
! --------------
! convention de signe spring : comprime:= +
set_[3]node_Phi
< var: ff_spring(.),
  fun:
    f_spring(inode)= - rk(inode)*(eta_move(inode) - eta_move(inode-1));
  var: ff_dump(.),
  fun:
    f_dump(inode)  = - rd(inode)*(eta_speed(inode) - eta_speed(inode-1));
>;
```

Comme on le voit sur l'exemple, le modèle attaché à une maille est générique et peut nécessiter la création de conditions limites particulières (transferts bidons à droite). Dans l'exemple, on a choisi mêmes nombres de mailles de type cellule ou transfert (c'est le [3] du set_node). Les noms de variables et fonctions sont libres mais doivent impérativement être différents. On dénomme une variable du maillage en une maille k par le symbole **(inode :k)**, où k est un chiffre ou un paramètre Fortran, ou encore une opération sur des variables définies. Le symbole **inode** est réservé. Des instruction Fortran habituelles peuvent être effectuées dans le bloc Mortran < > des set_xx_yy.

paramètres NGrid, MGrid Pour paramètrer les nombres de mailles, on utilisera d'une part l'instruction **parameter (NGrid=3,MGrid=3) ;** avec les deux formes **set_xxx** :

```
! node cells
! ----------
set_[NGrid]node_Eta
<
>;
! node transfers
! --------------
set_[MGrid]node_Phi
<
>;
```

La forme de l'instruction, comme le nom des variables **NGrid, MGrid** sont strictes.

écriture simplifiée

Lorsque l'utilisateur ne souhaite pas associer de nom à la fonction définissant le modèle correspondant à une ou plusieurs composantes, il peut remplacer une instruction :

```
  var: ff_dump(.),
  fun: f_dump(inode)  = - rd(inode)*(eta_speed(inode) - eta_speed(inode-1));
```

par :

```
eqn: ff_dump(.) = - rd(inode)*(eta_speed(inode) - eta_speed(inode-1));
```

Dans ce cas, la fonction prendra le nom de la variable précédée du caractère $: **$ff_dump(inode)**[1].

vérifications

Il est possible d'utiliser conjointement les instructions de base **f_set**, en prenant garde toutefois aux composantes définies par les **set_**. La correspondance avec les vecteurs et fonctions de base est donnée en impression d'exécution, où sont précisées aussi les mailles à droite (Dwn) et à gauche (Up) du maillage pour vérification :

```
  --------------- Informing on Dwn Eta definition ---------------
 Var-name,            Function-name, index in eta vector
        eta_pos_wall        deta_pos_wall  1 [
      eta_speed_wall      deta_speed_wall  2 [

  -------------- Informing on Eta Nodes definition --------------
 Var-name,     Function, k2index of (inode: 0 [ 1,...n_node ] n_node+1)
           eta_move           deta_move    1 [   3 ...   7 ]   9
          eta_speed          deta_speed    2 [   4 ...   8 ]  10

  ---------------- Informing on Up  Phi  definition -------------
 Var-name,             Function-name, index in ff vector
         ff_dummy_1           f_dummy_1 ]    7
         ff_dummy_2           f_dummy_2 ]    8
        ff_move_sum          f_move_sum ]    9
       ff_speed_sum         f_speed_sum ]   10
  ----------------------------------------------------

 -------------- Informing on Phi Nodes definition ---------------
 Var-name,     Function, k2index of (inode: 0 [ 1,...m_node ] m_node+1)
          ff_spring            f_spring   -1 [   1 ...   5 ]   7
            ff_dump              f_dump    0 [   2 ...   6 ]   8
  ----------------------------------------------------
```

En bref, on peut vérifier que les indices inférieurs nuls ou négatifs ne sont pas utilisés en limite d'équation de maille courante. Il est cependant conseillé de vérifier les cohérences avec les tableaux eta(.) et ff(.) dans "principal.f", c'est distrayant et aussi rassurant.

l'instruction mult_sum

L'impression précédente correspond en réalité à une modification de l'exemple donné, avec ajout de deux composantes de transfert :

```
set_up_Phi
< var:ff_dummy_1, fun: f_dummy_1=0.;
  var:ff_dummy_2, fun: f_dummy_2=0.;
  var: ff_move_sum, fun: f_move_sum = (mult_sum:eta_move(inode:[.i]),[i]=1,3);
  var: ff_speed_sum,
                fun: f_speed_sum =  ff_spring(inode:1)*eta_speed(inode:2);
>;
```

pour montrer l'utilisation de la macro **(mult_sum : ... ,[i]=ideb,ifin)**. où les arguments ideb et ifin sont des entiers chiffrés (ni paramètre ni variable FTN). Dans le cas présent, on obtient la somme explicite des trois termes. Cette macro s'utilise également dans tout contexte de définition de fonction, soit dans une **Fun_set :**, soit dans les blocs **set_xxx**, c'est-à-dire hors contexte Grid1D (cf in 2).

[1] L'initialisation éventuelle des transferts se fera en analogie avec celle des états, l'indice de boucle étant impérativement **inode**.

L'expression d'une mult_sum – obligatoirement entre parenthèses – peut être combinée à une expression plus complète, mais la forme de l'opérande doit rester simple du point de vue du nombre de parenthèses[1].

séquence DimEtaPhi

Nous donnons d'abord les détails de l'option d'écriture manuelle de la séquence **dimetaphi**, en rappelant qu'elle est optionnelle et choisie en fonction de **sel dimetaphi** dans la selseq.kumac. En mode de génération automatique, le [3] du set_node dans l'exemple précédent indique alors le nombre de mailles ; cette indication est ignorée en mode manuel.

Les dimentions des six groupes de variables doivent impérativement être données dans la séquence $DimEtaPhi (exemple non modifié) :

```
+SELF,IF=GRID1D.
! ++++++++++++++++++++++++++++++++++++++++++++++++++++++
! nodes parameters, and Limiting Conditions (Low and High)
! ++++++++++++++++++++++++++++++++++++++++++++++++++++++
      parameter (n_node=3,n_dwn=2,n_up=0,n_mult=2);
      parameter (m_node=3,m_ldwn=0,m_up=2,m_mult=2);
! ------------------------------------------------------
```

On a $n\_node = m\_node = 3$ mailles de type respectivement cellule et transfert, deux cellules de conditions limites à gauche (Dwn), aucune à droite (Up), le contraire en transferts. La multiplicité est le nombre de variables attachées à chaque maille cellule transfert (n_mult, m_mult), qui correspondra à une boucle **do io=1,n_node** en Fortran dans le code généré.

Pour que le code généré prenne en compte les spécificités du maillage, on doit lever le drapeau **Grid1D**. Ceci se fait dans la séquence **selseq.kumac** du répertoire oú se trouve la cmz file : l'instruction **sel Grid1D** lève ce drapeau (elle est commentée dans la selseq livrée avec les exos).

Version 102 Pour continuer à utiliser la séquence de dimentionnement manuel, il faut délectionner le drapeau dimetaphi dans la selseq.kumac (sel dimetaphi). Sinon, les instruction Fortran parameter seront automatiquement calculés et générés par Mini_ker, en prenant les nombres de mailles indiqués dans les set_[3]node.

runs avec beaucoup de variables

Il arrive que l'on dépasse la taille de mémoire vive de la machine, assez vite si on est en Grdi1D. Une erreur d'exécution se produit sans diagnostic clair, mais du genre **seg fault**[2]. Dans ce cas, faire précéder l'exécution de l'instruction Linux set ulimit unlimited.

---

[1] On peut sur commande demander plus, ou se reporter à une autre macro **mult_op** plus complète, cf paragraphe 3.1.

[2] qui se produit lorsqu'un sous-programme, bf onewt, oker réclame un tableau local (no save).

2 nouveaux transferts : les observables

La version **102** voit apparaître un nouveau vecteur (μ), déclaré ainsi :

```
set_probe
< var: p_sum, fun: f_sum= eta_pray + eta_pred;
>;
```

Il s'agit d'un transfert de type particulier, qui n'intervient pas dans les couplages avec les autres variables. Ils représentent typiquement un modèle d'appareil de mesure.

Leur intervention dans le code de **principal** est marquée par **ff(mp+[.j]**. Le fichier des sorties s'apelle **obs.data**, et il faut donner une dimension à **parameter (mobs=0)** dans la **dimetaphi** si nécessaire.

Bien sûr, la fonction de coût est sensible à ce vecteur dans l'adjoint.

Lorsque le filtre de Kalamn est utilisé (avec sel kalman dans la selseq), ces variables seront toutes utilisées – et elles seules – pour calculer le "vecteur innovation" (cf 7).

On peut utiliser la macro **mult_sum** dans tout contexte, comme ici :

```
Set_Probe
<
 eqn: mu_dist=(eta(1)+5.02832E+00)**2 + (eta(2)+7.43784E+00)**2
               + (eta(3)- 1.95888E+01)**2;
 var: sum_xyz, fun: somme_eta = (mult_sum: eta[i],[i]=1,3);
!* ou encore  var: sum_xyz, fun: somme_eta = mult_sum: eta[i],[i]=1,3;
!* ou bien    var: sum_xyz, fun: somme_eta = ftn_var*(mult_sum: eta[i],[i]=1,3);
>;
```

qui génère le code Fortran suivant, par exemple dans le calcul de cette fonction :

```
c ---*-------------------------------------------------------------I
10108 ff(mp+1)=(((eta(1)+5.02832E+00)**2+(eta(2)+7.43784E+00)**2+(eta(3)
     *-1.95888E+01)**2))
      ff(mp+2)=((eta(1)+eta(2)+eta(3)))
c ---*-------------------------------------------------------------I
```

Il est impératif de placer entre parenthèses l'expression concernée par la somme implicite.

102 Il est à présent plus commode d'utiliser des fonctions du genre précédent codées en Fortran. Tel est le cas de la fonction **quad_dist** qui permet dd calculer la distance quadratique entre deux points de l'espace des états ou des transferts. Exemple :

```
set_probe
< eqn: mu_dist = quad_dist(np,eta(1),eta_ref(1));
>;
```

l'observable **mu_dist** donne la distance quadratique entre l'état courant et un point de l'espace de phase pris comme référent. On peut y remplacer en argument **eta(1)** par un nom de variable symbolique (ou un transfer), mais celà n'évite pas d'avoir en tête le vecteur η (ou φ et sa traduction en tableau Fortran[1].

On utilise par exemple cette fonction pour évaluer la période (apparente[2]) d'un système.

[1]le principe de dérivation de ce type de fonction intégrale est donné dans la section 3.2.

[2]cette question débattue au Zircon de l'Alexandrienne a particulièrement intéressé Céline, à savoir : un système apparemment périodique est-il périodique ? C'est à ces hauteurs de questionnement qu'on reconnait l'unsecte.

2.1 Options graphiques

Les différents fichiers **.data** sont accessibles aux grapheurs courants. Ainsi, les fichiers peuvent être visualisés avec **gnuplot** en entrant :

```
plot "res.data" using 1:(n+1) for eta(n)
plot "tr.data" using 1:(n+1) for ff(n)
```

où n est un numéro de composante des vecteurs du TEF. Une premiere ligne indique les noms de variables associées aux colonnes dans le format gnuplot.

Pour les fans de **PAW**, logiciel graphique du CERN, Mini_ker génère les kumacs permettant la lecture des .data et leur structure en **n-tuples**. Se reporter au manuel PAW pour plus d'information. Les n-tuples ne sont cependant prêts à l'emploi que pour les vecteurs de dimension 10 au plus (y-compris la variable $time$). On édite ses kumacs pour les adapter au format souhaité - et on change leurs noms, car elles seront écrasées au prochain run.

Cette option est en réalité obsolete, la génération des kumacs n'ayant pas été remise à jour, faute d'utilisateur. On constate que non seulement la version ntuple de PAW est buggée, mais que gnuplot est d'un usage plus simple. Cependant, pour faire des histogrammes et des surfaces, PAW reste nettement supérieur. On conseille alors l'écriture d'un code Fortran pour générer les histogrammes nécessaires et ensuite les visualiser (surfaces incluses) dans PAW. On peut s'inspirer utilement de l'exemple fourni avec les Mini_ker sources dans **gains.f** *(avec go.xqt). La procédure à suivre dans PAW est automatiquement générée à l'éxécution dans la* **borel.kumac**.

2.2 Des sorties listing d'aide

On a déjà vu l'aide fournie pour le maillage. Pour les modèles non-maillés, on a un résumé des vecteurs en listing d'exécution, comme par exemple pour Lorenz-63[1] :

```
 ---------------- Informing on Phi definition -----------------
   Var-name,           Function-name,       index in ff vector
           ff_courant_L          $ff_courant_L  1
              ff_T_czcx             $ff_T_czcx  2
           ff_Psi_Tczcx          $ff_Psi_Tczcx  3
             ff_Psi_Tsz            $ff_Psi_Tsz  4
 ----------------------------------------------------

 --------------- Informing on Eta definition ------------------
  Var-name,           Function-name,       index in eta vector
          eta_courant_L              deta_conv  1
             eta_T_czcx         deta_energy_zx  2
               eta_T_sz          deta_energy_z  3
 ----------------------------------------------------
```

Un résumé des équations est également disponible dans le fichier **Model.hlp**, comme le suivant, pour le même Lorenz :

```
==================== set_Phi
 1 ff_courant_L  $ff_courant_L   eta_courant_L
 2 ff_T_czcx     $ff_T_czcx      eta_T_czcx
 3 ff_Psi_Tczcx  $ff_Psi_Tczcx   eta_T_czcx*eta_courant_L
 4 ff_Psi_Tsz    $ff_Psi_Tsz     eta_T_sz*eta_courant_L

==================== set_Eta
 1 eta_courant_L deta_conv       pi_prandtl*ff_T_czcx-pi_prandtl*eta_courant_L
 2 eta_T_czcx    deta_energy_zx  -ff_Psi_Tsz+pi_Rayleigh_ratio*ff_courant_L-eta_T_czcx
 3 eta_T_sz      deta_energy_z   ff_Psi_Tczcx-pi_wave_nb_ratio*eta_T_sz
```

les fonctions définies par des **F_set** ne figurent dans ce fichier que si elles sont déclarées par **Fun_set**. On peut encore abonder l'information grâce à la macro **Aux_fun**, de même format, qui ne génère rien pour Mini_ker mais est écrite dans **Model.hlp**. Exemples :

```
Fun_set no_w_neg_0 = (1.-ostepin(-w(0)-dt*(adv_w(0)+buoy_w(0)+sol_w(0))+dm2,2.*dm2))
Aux_fun dm2 = 0.01;
```

Dans la mesure du possible toutes les valeurs numériques sont repérées par un nom de variable. Ainsi, en listing d'exec :

```
Gamma :-8.19100E-02-1.42151E-01 3.87150E-02
        eta_courant eta_T_czcx  eta_T_sz
      ------------------------------------------------
Omega : 0.00000E+00 0.00000E+00 0.00000E+00 0.00000E+00
        ff_courant_ ff_T_czcx   ff_Psi_Tczc ff_Psi_Tsz
      ------------------------------------------------
```

pour les vecteurs connus par exemple, ou pour la matrice de couplage :

[1] fourni, cf **lorhcl**.

```
 >ker : Matrice de couplage       4 4 4 4
ff_courant_L Raw(  1,j=1,  4):   1.000     -9.9010E-03   0.000       0.000
ff_T_czcx    Raw(  2,j=1,  4): -2.7972E-02   1.000       0.000      9.9900E-04
ff_Psi_Tczcx Raw(  3,j=1,  4):  0.1605      9.7359E-02   1.000     -5.7321E-03
ff_Psi_Tsz   Raw(  4,j=1,  4):   0.000     -0.1376      5.7225E-03   1.000
             Var-Name          ff_courant_ ff_T_czcx   ff_Psi_Tczc ff_Psi_Tsz
             ----------------------------------------------------------
```

comme pour les Jacobiennes et autres matrices de Mini_ker, de même encore comme déjà indiqué en tête des fichiers **data**.

Les grosses matrices bénéficient d'un traitement particulier : elles ne sont pas données en tableaux de chiffres mais sous la forme de "scatter plots". Nous commentons ici un exemple extrait d'UrbaGus :[1]

```
 ------ TEF Matrices and vectors in ker ---------
 ------------------------------------------------
  A_matrix in ker                 48 48 48 48       >>>   2229 zero values  <<<
 [MIN:-0.4252    (32,33) (@zyxwvutsrqpi, .1PQRSTUVWXYZ*)  214.0    (43,43):MAX]
    123456789012345678901234567890123456789012345678
  1|P                                               || hous(1st)
  2|,1,                                             || rent(1st)
  3|  1                                             || densite(1st)
  4|   P                                            || hous(.)
  5|   ,1,                                          || rent(.)
  6|     1                                          || densite(.)
  7|      P                                         || hous(.)
  8|      ,1,                                       || rent(.)
  9|        1                                       || densite(.)
 10|         P                                      || hous(.)
 11|         ,1,                                    || rent(.)
 12|           1                                    || densite(.)
tableau tronqué
 37|                                    P           || hous(.)
 38|                                     1,         || rent(.)
 39|                                      1         || densite(.)
 40|                                       P        || hous(.)
 41|                                        1,      || rent(.)
 42|                                         1      || densite(.)
 43|                                          *     || hous(last)
 44|                                          ,1,   || rent(last)
 45|                                            1   || densite(last)
 46|                                             1  || hous_dum
 47|                                              1 || rent_dum
 48|                                               1|| densite_dum
    123456789012345678901234567890123456789012345678
    .    .    .    .    .    .    .    .    .    . densite_dum
    .    .    .    .    .    .    .    .    .    .rent_dum
    .    .    .    .    .    .    .    .    .    hous_dum
    .    .    .    .    .    .    .    .    .   densite(last)
    .    .    .    .    .    .    .    .    .  rent(last)
    .    .    .    .    .    .    .    .    . hous(last)
```

[1]thèse de Gus)

```
sortie tronquée
    .    .  densite(.)
    .    . rent(.)
    .    .hous(.)
    .    densite(.)
    .   rent(.)
    .  hous(.)
    . densite(1st)
    .rent(1st)
    hous(1st)
```

Une première ligne donne la taille de la matrice (deux indices par dimention : le premier la partie imprimée, la deuxième la taille réelle de la matrice dans sa "Dimension" Fortran). On donne aussi le nombre réréré de coefficients strictement nuls.

La deuxième ligne indique le code alphabétique corespondant aux valeurs des coefficients de la matrice, encadrés par les valeurs min et max référées. Ici, la plus petite valeur est de -0.4252 en position $(32, 33)$ (@ dans le tableau), le max $= 214.$ en position $(43, 43)$ (* dans le tableau).

Les minuscules codent des valeurs négatives, les majuscules des positives. Enfin, pour respecter le TEF, on code systématiquement la valeur 1 par 1 (et -1 par i), l'intervalle max par $*$ (et @ le min si négatif). Enfin la valeur stricte 0 est codée par un blanc, les valeurs inférieures à 1. en valeur absolues sont codées par "." lorsqu'elles sont positives, et par "," si négatives. Entre 1. et max (resp. -1. et min si nécessaire) on code 12 intervalles de P,Q,R,..., à *.

On me dit "c'est bien", mais "pourquoi les poils ?". Les poils servent à repérer les variables associées à chaque coefficient. Et comme ici dans le cas de modèle maillé, la variable est indicée par (1st) et (last), pour la première et dernière maille, les mailles intermédiaires par (.).

Test de linéarité sur φ

Lors de la fin du pas de temps, qui fournit les transferts, un test numérique est effectué par comparaison de $\varphi = f(\eta(t - dt) + d\varphi$ (**ffl**), avec $d\varphi$ donné par la résolution de $d\eta, d\varphi$ (dans **ker**) et $\varphi = f(\eta, \varphi, t)$ (**ff**) :

```
! ==========================================================
! 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*,'*** pb linearite : res > TOL_FFL a istep',istep,res,' > ',TOL_FFL;
    do io=1,m < z_pr: io,ff(io),ff(io)-ffl(io); >;
  >;
>;
```

Comme on le voit, il s'agit d'un critère portant sur la somme des écarts quadratiques entre valeurs relatives (ou non). Ce test n'englobe ainsi que les non-linéarités des modèles de transfert. On peut utiliser la valeur de **res** dans **ZSTEER** pour piloter la valeur de **dt**, voire, revenir au pas de temps précédent. Les plus grands écarts de la liste sont signalés par $<<<$.

Il est aussi possible de définir une valeur du critère **TOL_FFL** dans **ZINIT** différente du défaut fixé à 10^{-3} indépendamment de la précision Fortran.

3 Principes utiles à la programmation

3.1 Comment rendre un modèle continu

[1]On traite d'un problème lié à la continuité des fonctions et de leurs dérivées partielles. Comme exemple, on peut avoir une fonction step de la variable, comme dans les modèles avec changement de phase, avec des codes en Fortran comprenant un **IF** sur la variable de température. Dans ce cas, on régularise le problème en remplaçant la step par une "smooth step". On peut noter à ce sujet qu'il s'agit d'une prise en compte implicite de l'échelle spatio-temporelle. A une échelle macroscopique, la fonction step est juste un non-sens.

On peut avantageusement utiliser la fonction **Heavyside**[2] de la //mini_ker/mathlib :

```
Delta = -1."K";
A_Ice =  heavyside("in:" (T_K-Tf), Delta, "out:" dAIce_dT);
```

dans l'exemple, Tf est la température de fusion de la glace, A_ice donne la fraction englacée d'un volume d'eau à la température T_k. La fonction smooth-step est une quasi tangente hyperbolique fonction de la variable adimentionnelle x/Δ), normalisée de 0 à 1, avec une pente maximum de 2.5, voire figure 1. Pour avoir la dérivée, il suffit d'ajouter (en tête de

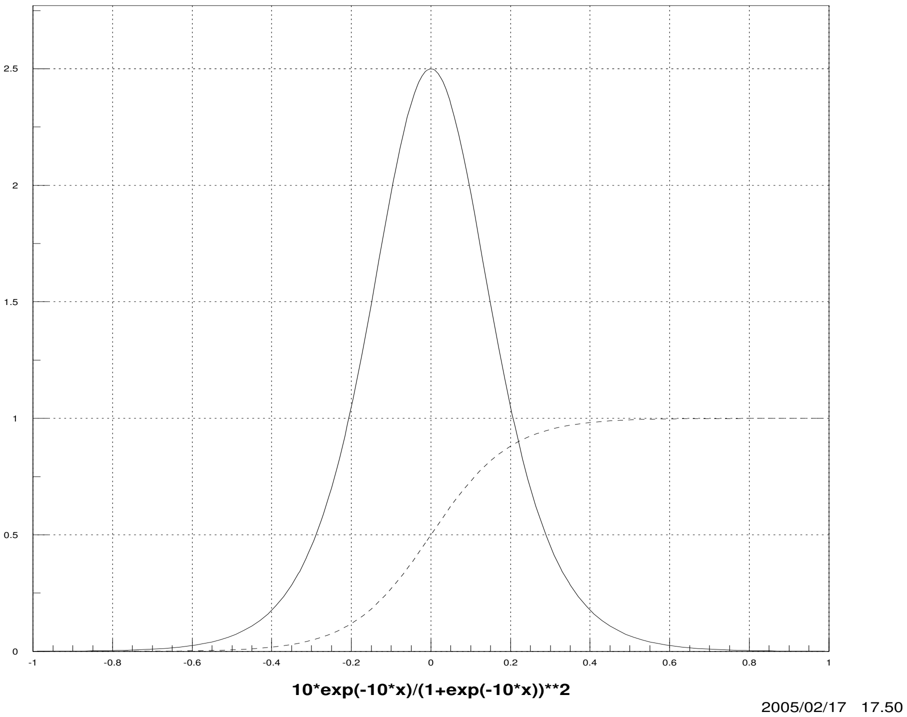


FIG. 1 – La fonction "domptée" de Heaviside et sa dérivée pour une variable adimentionnée

zinit) la règle de dérivation :

```
&'(HEAVYSIDE(#,#,#))(/#)' = '((#1)(/#4)*HEAVYDELTA(#1,#2,#3))'
```

(cette règle fait à présent partie des macros de Mini_ker, de même la fonction et son point d'entrée (HeavyDelta).

Un autre type de problème consiste en la traduction de $var = min(f(x), g(x))$. Dans ce cas, on ne veut pas dériver la smooth-step, et on écrira simplement

```
var = HeavySide(f(x)-g(x),Delta,dum)*g(x)  + (1.-HeavySide(f(x)-g(x),Delta,dum)*f(x);
```

[1]cette section concerne donc aussi les instructions **set_eta¡ et semblables.**

[2]je précise pour éviter les propagations malencontreuses que "Heavyside" est un joke, pour dire une fonction de Heaviside "lourde", c'est-à-dire à la fois inerte et miraculeuse.

ou de manière équivalente :

```
var = HeavySide(f(x)-g(x),Delta,dum)*g(x)  + HeavySide(g(x)-f(x),-Delta,dum)*f(x);
```

avertissement La valeur de Delta en argument est importante et doit être soigneusement choise, car de cette valeur dépend la valeur de la dérivée partielle qui interviend dans les Jacobiennes.

ATTENTION ! Il est fortement déconseillé de donner en argument des valeurs numériques. En effet, avec passage en double précision, les chiffres restent en simple[1] alors qu'ils seront vus en double dans la fonction : erreur garantie.

Autre fonction de transition

La Heavyside est par construction symétrique autour de zéro. Il est de nombreux cas où on veut une fonction permettant une transition strictement entre zéro et **Delta**, comme par exemple deux fonctions prenant chacune en compte le cas de l'argument positife ou négatif. C'est le rôle de la fonction **oStepin**, avec la définition :

ostepin(x,Delta)= $\dfrac{x}{\Delta} - \sin(\dfrac{2\pi x}{\Delta})/2\pi$,

strictement nulle pour les valeurs négatives de x, égale à l'unité dès que $x/\Delta \geq 1$), et suivant la fonction sin entre 0 et 1. Pente max de 2. pour $\frac{x}{\Delta} = 0.5, \Delta = 1.$[2].

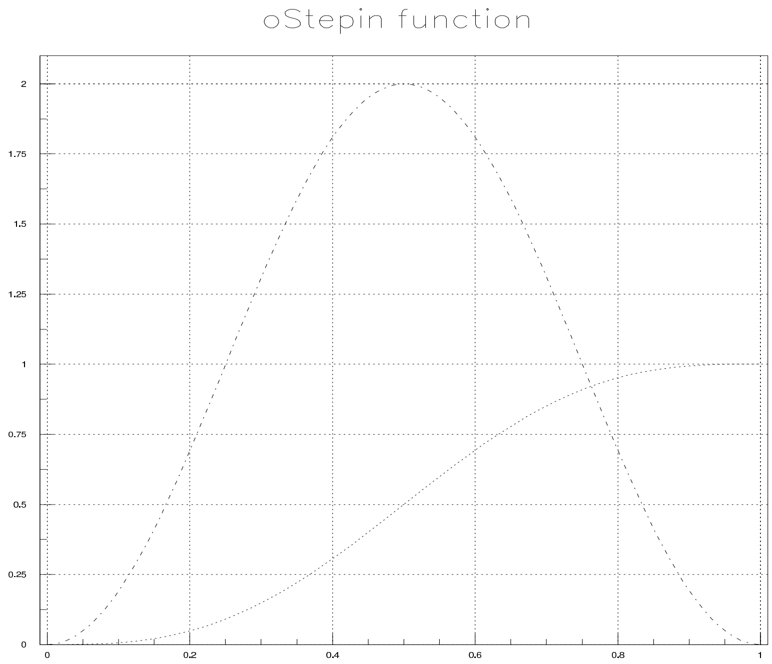


FIG. 2 – Une fonction step et sa dérivée pour une variable adimentionnée x/Δ (dérivée pour $\Delta = 1$).

La règle de dérivation utilise la fonction associée **oDeltin** :

```
&'(OSTEPIN(#,#))(/#)' = '((#1)(/#3)*ODELTIN(#1,#2))'
```

avec la fonction dérivée $= (1 - cos\frac{2\pi x}{\Delta})/\Delta$.

Ainsi en zéro la fonction et sa dérivée est nulle.

[1]sauf à préciser qu'ils sont en double, comme 1.D0, mais alors ils restent en double si on repasse en simple.

[2]prendre garde au fait que la fonction ne dépend que de $\frac{x}{\Delta}$, mais la dérivée change avec Δ.

3.2 Règles de dérivation

Se souvenir que les règles de la dérivation formelle correspondent à un tableau de macros **Mortran** dans la séquence **P=INCLUDE, D=$Derive_mac** de Mini_ker. En cas de doute, il faut vérifier que les fonctions mathématiques utilisées dans les modèles y sont traitées. Il est par ailleurs très simple d'en ajouter en analogie avec celles qui y sont.

Il est par exemple est assez courant d'avoir à utiliser la fonction intrinsèque Fortran **abs()**. Il est bon de vérifier que la règle de dérivation est, soit dans la séquence, soit l'ajouter dans votre ZINIT :

```
 &'(ABS(#))(/#)' = '((#1)(/#2)*SIGN(1.,#1))'
!                                "derivation de la fonction valeur absolue"
```

De manière générale, il est souvent plus opportun de concevoir une macro supplémentaire que de jouer au Fortran (du genre sqrt($f()^2$) pour avoir une abs). Dans ce cas, il est impératif d'utiliser les noms génériques de fonctions implicites, ceci pour pouvoir passer en double précision sans souci (ex : **dsin()** est interdit, c'est **sin()** qu'il faut utiliser).

On peut aussi entrer ses valeurs numériques systématiquement en double, ce qui n'aura aucune conséquence en simple. Exemple :

```
pi = acos(-1.D0);
```

La variable déclarée **real pi** passera de 7 à 16 chiffres significatifs automatiquement en double[1]. Celà signifie aussi qu'il est conseillé de déclarer **real** – et non pas **dimension** ces variables là ou tableaux (souvent des paramètres du modèle) ; ça se fait en tête du **ZINIT**. *On utilisera utilement l'option selseq :* **sel none** *pour s'assurer que toutes les variables ont bien été déclarées (cette option provoque la compilation de* **principal** *avec la règle* **implicit none***).*

ATTENTION Cette règle ne concerne que les fonctions implicites Fortran. Pour les autres fonctions, le type d'un argument sous forme de chiffre conserve sa précision de déclaration : 1. reste sur quatre octets en double, 1.D0 reste en huit octets en simple précision. NE PAS utilisez d'arguments chiffrés en règle générale (cf fonction **Heavyside**).

IMPORTANT : Ne jamais utiliser dans une équation de valeurs numériques sous la forme FTN "1.2E-3", car la dérivation symbolique ne l'interprète pas correctement[2]. Ainsi, l'entrée de eqn : var_eta1 = 1.2E-3*(var_ff1-var_eta2) conduira à des dérivées égales à ± 3 ! Un principe général de bonne programmation est d'utiliser des variables fortran pour ranger ces valeurs.

Dérivation de puissance fonctionnelle de fonction.

La dérivation partielle par rapport à un exposant fonctionnel n'est pas fiable ($d_y g(x,y)^{f(y)}$). On doit la remplacer par **power(g,f)**, ou si l'on préfère, écrire $exp(f(y)*log(g(x,y)))$. La dérivation de la fonction **power** est prise sour la forme suivante :

$$\partial_x f^g = g f^{g-1} \partial_x f + f^g \log f \partial_x g = f^{g-1}(g \partial_x f + f \partial_x g)$$

et se trouve dans le tableau des règles de dérivation **$DERIVE_MAC**.

3.3 Modèle de type face, interface

On entend par là un code fournissant des grandeurs indépendantes du modèle de simulation. Par exemple, un code fournissant des grandeurs météo pour calculer des sources aux parois extérieures, des flux solaires etc.

[1] une macro transforme en effet **real** en **double precision** quand on sélectionne le flag **double** dans Mini_ker.

[2] "bug" signalé par stb en Mai 2007

La meilleure manière est de coder une fonction fortran qui fasse l'interface informatique entre le code externe (lecture de fichiers, calculs etc). Dans la mesure où il fournit des variables indépendantes, les dérivées partielles de la détyermination des Jacobiennes sont nulles, il n'ay a ainsi pas à fournir de règles de dérivation de la fonction.

On aura donc un codage du genre :

```
set_phi
<  eqn:temp_paroi = Code_temp(time, orientation);
>;
```

avec Code_temp() qui va faire tout ce qui est nécessaire pour fournir la température de l'air extérieur vue par une paroi en fonction de la date, de son orientation, etc.

Attention Il peut être prévu d'étudier les sensibilités à une oriention de paroi, auquel cas la fonction devra calculer en plus sa dérivée par rapport à cette grandeur. On écrira alors dans le tableau des macros **Derive_Mac** une règle comme suit :

```
&'(Code_temp(#,#))(/#)' = '((#1)(/#3)*Dcode_Temp(#1,#2))'
```

avec un point d'entrée ajouté à la fonction Code_temp pour fournir la valeur de cette dérivée. La section suivante complète utilement ces principes.

3.4 Jacobiennes numériques

Certains modèles ne peuvent être analytiquement formulés et exigent une procédure numérique. C'est le cas des calculs radiatifs GLO basé sur la procéduree de Malkmus. Dans de tels cas, les macros de dérivation ne peuvent opérer symboliquement sur une expression analytique du modèle, et on leur substitue un calcul numérique., qui doit être ajouté au modèle de détermination numérique des grandeurs – ici des flux radiatifs.

Dans l'exemple suivant extrait de "ClimSI"[1], il reste dans ZINIT une expression analytique représentant le modèle numérique complet linéarisé localement par rapport à un état de référence (les variables X_ref) (état qui change en cours de simulation). Les coefficients de linéraisation (flux_xxx(.)) sont calculés par le modèle numérique.

Quand donc ? Il y a une subtilité : l'équation du premier flux (LW_oce) possède la particularité par rapport aux deux autres de faire un appel à une fonction Fortran (ici Flux_Rad(...)) qui fait l'interface entre Mini_ker et le modèle numérique :

```
set_phi
<
 * * * * *
! ==============================================================
! Long wave radiation flux between Ocean-Atmosphere and Space
!   (3 variables : Phi_Oce and Phi_Atm and Phi_Str)      TRAD
! ==============================================================
!*Equ:       Phi_Oce = -sigma(SST^4-T^4)
!*Equ:       Phi_Atm = sigma(SST^4-T^4-(T+Gammat*haut(Q))^4)
!*Equ:     and Malkmus narrow band model from Cherkaoui et al.

! -------------------------------------------------------------------
  eqn:  flux_LW_ocean = Flux_Rad(
           T_ocean_eta,T_tro_surf_phi,rLapse_tro,T_strato_eta,rLapse_str,
           HumRel_phi,xCO2,rN_B,rN_M,rN_H,IDO,time,istep,Zprint,ZnewFlux
                       ) + flux_oce_ref
                         + flux_oce(2)*(T_ocean_eta-Tocean_ref)
```

[1] thèse de StepH.

```
                         + flux_oce(3)*(T_tro_surf_phi-Tsurfair_ref)
                         + flux_oce(4)*(T_strato_eta-Tstrato_ref)
                         + flux_oce(5)*(HumRel_phi-HumRel_ref)
                         + flux_oce(6)*(xCO2-xCO2_ref)
                         + flux_oce(7)*(rN_B-rN_B_ref)
                         + flux_oce(8)*(rN_M-rN_M_ref)
                         + flux_oce(9)*(rN_H-rN_H_ref);



  eqn:  flux_LW_tropo = flux_trp_ref
                         + flux_trp(2)*(T_ocean_eta-Tocean_ref)
                         + flux_trp(3)*(T_tro_surf_phi-Tsurfair_ref)
                         + flux_trp(4)*(T_strato_eta-Tstrato_ref)
                         + flux_trp(5)*(HumRel_phi-HumRel_ref)
                         + flux_trp(6)*(xCO2-xCO2_ref)
                         + flux_trp(7)*(rN_B-rN_B_ref)
                         + flux_trp(8)*(rN_M-rN_M_ref)
                         + flux_trp(9)*(rN_H-rN_H_ref);

  eqn:  flux_LW_strato = flux_str_ref
                         + flux_str(2)*(T_ocean_eta-Tocean_ref)
                         + flux_str(3)*(T_tro_surf_phi-Tsurfair_ref)
                         + flux_str(4)*(T_strato_eta-Tstrato_ref)
                         + flux_str(5)*(HumRel_phi-HumRel_ref)
                         + flux_str(6)*(xCO2-xCO2_ref)
                         + flux_str(7)*(rN_B-rN_B_ref)
                         + flux_str(8)*(rN_M-rN_M_ref)
                         + flux_str(9)*(rN_H-rN_H_ref);
! -----------------------------------------------------------------

>;"end set_Phi"
```

Les trois expressions ci-dessus seront toujours appelées dans le même ordre, et ainsi, l'appel à la fonction Flux_Rad aura pour conséquence un calcul complet des flux par le modèle numérique, la détermination de l'état courant de référence, et les coefficients du modèle linéarisé. Ces coefficients seront en général déterminés par accroissements finis. Au modèle numérique encore d'optimiser les calculs, en rappelant le modèle complet seulement lorsque l'état de référence aura significativement changé. Dans l'exemple présenté, pour diminuer le nombre d'arguments, les variables de référence ainsi que les coefficients sont mis en **common** au début de ZINIT :

```
! ===== Interface to Flux_rad subroutine ==============================
 Logical ZnewFlux;
 real FLUX(4,4,9), Flux_oce(9), Flux_trp(9), Flux_str(9);
! Referenced Values for linearisation
 real Tocean_ref, Tsurfair_ref, Tstrato_ref, HumRel_ref, xCO2_ref,
               flux_oce_ref, flux_trp_ref, flux_str_ref,
           rN_B_ref, rN_M_ref, rN_H_ref, rLapse_tro_ref, rLapse_str_ref;

 common/fluxrad/Flux_oce, Flux_trp, Flux_str,
               flux_oce_ref, flux_trp_ref, flux_str_ref,
          Tocean_ref, Tsurfair_ref, Tstrato_ref, HumRel_ref, xCO2_ref,
```

```
          rN_B_ref, rN_M_ref, rN_H_ref, rLapse_tro_ref, rLapse_str_ref;
! ======================================================================
```

3.5 Fonctions de grille

Il est courant d'avoir à calculer une fonction globale portant sur une variable de grille, comme une moyenne spatiale, une variance, etc. Une manière classique de le faire est de creer une variable en chaque maille effectuant la somme du terme de maille sur la maille et sa voisine (de droite ou de gauche). Alors, la derniere maille contiend la fonction sommée sur les mailles et peut être traitée pour fournir le résultat final avec une ultime fonction de bord. Cette procédure consomme $n+1$ variables. ; On a encore la fonction mult_sum, mais celle ci genere des calculs de dérivation en Fortran dont la longueur fait facilement exploser Mortran.

Il est préférable de faire usage d'une librairie de fonctions de grille sobres. Deux premières fonctions effectuent la moyenne et la variance d'une variable sur n mailles à partir d'une variable explicitée, comme ça :

```
var: util_moy, fun: avrg_phi(12,util2(inode:1));
```

qui est transformée par une macro en :

```
 ...    avrg_node(12,util2(inode:1),j,ff(1),m_mult);
```

qui renvoie à la fonction Fortran. Son point d'entrée **davrg_node** fournit la dérivée par rapport à ff(j) (l'argument **j** est bidon). Le principe est de récupérer les indices du tableau via la fonction **locf** du CERN[1].

La variance est fournie par **vari_eta()** et sa sœur **vari_phi** ; elle est donnée par $\frac{1}{n}\sqrt{\sum_{i=1}^{n}(\zeta_i - \overline{\zeta})^2}$, où - indique la moyenne de la variable de maille ζ :

```
var: util_quad, fun: vari_phi(12,densite(inode:1));
```

Le point d'entrée de la dérivée est **dvari_node**.

Fonctions de grille pondérées

Les mêmes fonctions que précédemment dont l'argument est un produit de variables du TEF, pour la moyenne pondérée, dans cet exemple issu de **Urbadyn** avec une utilité en variable de transfert pondérée par une densité (d'habitants en l'occurence) qui est une variable d'état :

```
any = avrg_poids(12,Eta:densite(inode:1),Phi:util2(inode:1));
!                   ---                  ---
any = avrg_poids(12,Phi:util2(inode:1),Eta:densite(inode:1));
any = avrg_poids(12,Phi:util2(inode:1),Phi:util2(inode:1));
any = avrg_poids(12,Eta:densite(inode:1),Eta:densite(inode:1));
;
j=56; "choisi pour correspondre à un indice de ff(.) <-> util2(inode:2)"
any = F_D(avrg_poids(12,Phi:util2(inode:1),Phi:util2(inode:1)))(/ff(j));
;
i=13;
any = F_D(avrg_poids(12,Eta:densite(inode:1),Eta:densite(inode:1)))(/eta(i));
```

Les points d'entrée sont **avrg_two, davrg_two1, davrg_two2** car il faut pouvoir dériver suivant l'une et l'autre des deux variables de grille.

[1] Oui, ça marche aussi en double précision.

Pour la variance pondérée $\frac{1}{n}\sqrt{\sum_{i=1}^{n}(\xi_i\zeta_i - \overline{\xi\zeta})^2}$:

```
any = vari_poids(12,Eta:densite(inode:1),Phi:util2(inode:1));
any = vari_poids(12,Phi:util2(inode:1),Phi:util2(inode:1));
any = vari_poids(12,Eta:densite(inode:1),Eta:densite(inode:1));
;
j=56;
any = F_D(vari_poids(12,Phi:util2(inode:1),Phi:util2(inode:1)))(/ff(j));
!*** Code Genere
!       any=(DAVRG_TWO1(12,ff(k2j_12(1)),ff(k2j_12(1)),j,FF(1),FF(1),M_MUL
!      *T,M_MULT)+DAVRG_TWO2(12,ff(k2j_12(1)),ff(k2j_12(1)),j,FF(1),FF(1),
!      *M_MULT,M_MULT))
;
i=13;
any = F_D(vari_poids(12,Eta:densite(inode:1),Eta:densite(inode:1)))(/eta(i));
```

Les points d'entrée sont **vari_two, dvari_two1, dvari_two2**. (cf les routines dans **mathlib** et les règles de dérivation dans **derive_mac**.

Attention : Il n'y a pas de diagnostic d'erreur concernant la nature **Eta, Phi** des variables.

Indice de Gini

La fonction **Gini_index** a été introduite dans Mini_ker par François Gusdorf. On considère une population N partageant une utilité commune (u), et la courbe de O. Lorenz associée donnant, sur p groupes de population ordonnés en u croissant, la part d'utilité totale $N\overline{u}$ que partage la population cumulée $\int_0^u N(v)dv, \quad v \leq u$.

Le coefficient de Gini donne un indice d'inégalité de partage du gâteau sous forme d'une intégrale : la surface comprise entre la première bissectrice (égalité parfaite) et la courbe de Lorenz, affectée d'un facteur 2 (ainsi, le Gini varie entre 0 et 1). Exemple tiré de l'UrbaGus :

```
set_Probe
<  eqn: u_quad = vari_phi(n_node,util2(inode:1));
   eqn: gini_dx = Gini_index(n_node,ETA:densite(inode:1),PHI:util2(inode:1));
>;
```

dans lequel la population est en premier argument et l'utilité (ou un revenu) en deuxième.

Nous ne saurions éviter de poser la question de l'intérêt de cet indice issu de l'éconnomie pour caractériser un état physique : est-ce que les diphasiques de l'unsecte pourraient brainstormer là-dessus ?

3.6 Facilités pour gnuplot

Surfaces dans gnuplot

Le format des fichiers **.data** ne permet pas simplement de faire des graphes 2D du genre profil x temps. On dispose d'un petit éxécutable qui permet de le faire aisément. Usage :

```
 splot "<1dgrid res.data 4 12 5" u 1:2:3 w l
 ou encore
 splot "<grid1d res.data 4 12 5" u 1:2:3 w l
```

en lieu et place de la lecture du fichier ici res.data, grid1d est un éxécutable dont on montre l'installation plus bas, qui va extraire de ce fichier les valeurs indiquées par les arguments qui suivent. Le premier, 4, est le numéro de la première colonne où se trouvent les valeurs de la grandeur sélectionnée ; 12 est le nombre de mailles — qui pouura englober bien sùr les conditions limites (up et/ou dwn) ; 5 est le nombre d'équations par mailles. Ces informations se retrouvent dans le Model.hlp habituel, mais il s'agit du numéro de colonne du fichier, comme à l'habitude pour gnuplot. Le résultat est un profil en fonction de tous les instants de la simulation, en filaire. Avec la version 4 de gnuplot, "set pm3d" trace des surfaces en couleur.

On peut aisément récupérer le nom de la grandeur en faisant "

```
gnuplot> !1dgrid res.data 4 12 5|tail -1
 qui donne
gnuplot> title "densite(1st)"
 on trace alors par
gnuplot> splot "<1dgrid res.data 4 12 5" u 1:2:3 title "densite(.)" w l
```

par exemple.

Arguments supplémentaires :

```
 splot "<1dgrid res.data 4 12 5 100 1000 10" u 1:2:3 w l
```

avec 100 le premier istep demandé, 1000 le nombre à tracer, et 10 de dix en dix. Les défaults sont en principe corrects.

Exemple avec les masselottes :

```
gnuplot> set xlabel "t"
gnuplot> set ylabel "maille"
gnuplot> set zlabel "grandeur"
gnuplot> !1dgrid res.data 2 4 2 1 1|tail -1
gnuplot> set title "eta_pos_wall"
!
gnuplot> splot "<1dgrid res.data 2 4 2 1 500 40" u 1:2:3
                                                       title "eta_pos(.)" w l
gnuplot> replot "<1dgrid res.data 3 4 2 1 500 40" u 1:2:($3-0.02)
                                                      title "eta_speed(.)" w l
```

on obtiend en filaire :

Installation :

Récupérer timap.f dans /pub/alain, le compliler avec, par exemple,

```
 f77 -fno-backslash -o timap.exe timap.f
```

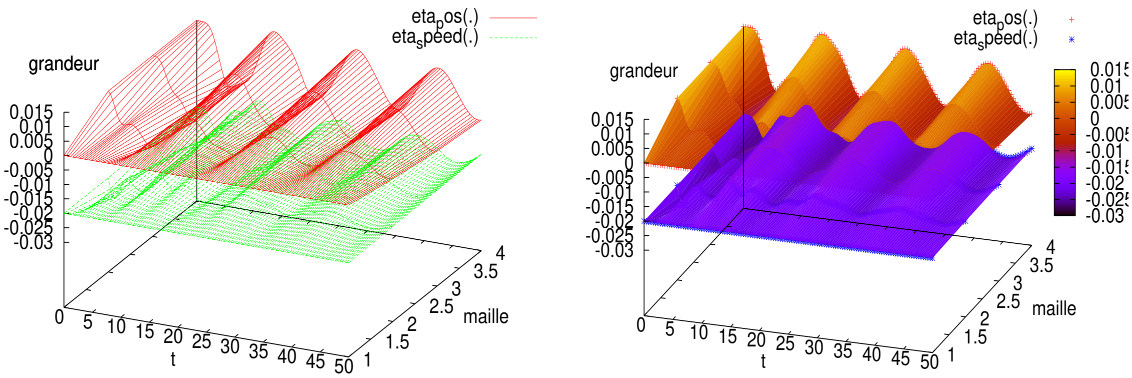
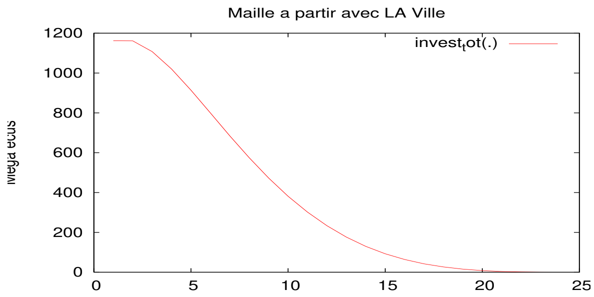


FIG. 3 – Exemples de sortie 1dgrid en filaire et avec pm3d.



Profil simple
Obtenir un seul profil peut être extrêmement intéressant pour réinitialiser le run après modification du nombre de mailles.
Pour sortir le 50ième pas de temps, il suffit de faire un simple plot pour le résultat ci-contre.

FIG. 4 – plot "<grid1d tr.data 42 24 22 50 1" u 2 :3 title "invest_tot(.)" w l

il faut recopier ce .exe dans /usr/bin (ou local/bin suivant vos configs). Il ne reste plus qu'à créer un ou plusieurs liens pour l'appeler autrement (1dgrid, grid1d...).

Version avec toutes les variables

La version **timapx** donne en sortie toutes les variables, qui seront alors accessibles dans gnuplot à partir de la colonne 3, les deux premières étants réservées au temps et au numéro de maille. On peut ainsi effectuer des opérations entre variables ; exemple :

```
splot "<1dgridx res.data 5 12" u 1:2:3 w l
(identique au précédent) ou encore
splot "<grid1dx res.data 5 12" u 1:2:($3+$4) w l
```

effectuant la somme des deux premières variables.

Comme on le voit, il y a un argument de moins, le premier étant directement l'incrément (la première colonne sera toujours la deuxième du fichier à lire). Les trois arguments de gestion du temps sont inchangés. L'installation est semblable à celle de **timap**.

Histogrammes dans gnuplot

Un autre éxécutable **histo.f**, qui s'installe comme les précédents, permet de faire des histogrammes à partir des fichiers colonne **.data**.

Usage :

```
plot "<hist dres.data 12 100 1. 20. 1 100 5 " u 1:2 w hist
```

effectue un histogramme sur la 12ème colonne du fichier **dres.data** avec 100 canaux de 1. à 20. en prenant 100 données, du premier incrément et en allant de 5 en 5.

4 Principes de la résolution

Dans la boucle temporelle, **principal** résout d'abord l'équation matricielle de transfert. Le calcul de la matrice D est effectué en utilisant les dérivées partielles $\delta_\varphi \varphi$ symboliquement construites par le passage de Mortran (le lecteur intéressé par la technique peut se reporter à la première annexe au présent document).

Le système TEF se construit de manière analogue par calcul de toutes les matrices Jacobiennes du système :

$$\begin{aligned} H = \partial_\eta g \quad &; \quad B_b = \partial_\varphi g; \\ C^\dagger = \partial_\eta f \quad &; \quad D = \partial_\varphi f; \end{aligned} \tag{4}$$

Deux matrices sont alors construites. Une première correspondant au Système Linéaire Tangent (SLT) :

$$\begin{bmatrix} \partial_t \cdot -H & -B_b \\ -C^\dagger & 1-D \end{bmatrix} \begin{bmatrix} \Delta\eta \\ \Delta\varphi \end{bmatrix} = \begin{bmatrix} u^\eta \\ u^\varphi \end{bmatrix} \tag{5}$$

avec les vecteurs u de sources ou forçages perturbants. On élimine les transferts de ce système, pour obtenir la matrice d'avance des systèmes linéaires d'état classiques (*aspha* dans le code). Elle est sortie dans le fichier **aspha.data** pour analyse post-run (cf SLT, SLTC), et ne sert pas à la résolution (comme ce fichier pet-être trés volumineux, on active sa sortie avec l'option **sel aspha** dans la selseq.kumac).

Le système est alors discrétisé en temps selon un schéma de Crank Nicolson, pour la résolution numérique du système à chaque pas de temps. Cela se fait par un appel à la routine **oker**, qui élimine les cellules pour former la matrice de couplage (tableau **couplage(.)** dans le code) entre tous les transferts du système, résout le système linéaire réduit obtenu, calcule les variations $\delta\varphi$ puis de $\delta\eta$. L'état est alors avancé dans "principal" et on peut résoudre le système implicite des transferts ; fin du pas de temps.

La matrice de couplage n'est pas sortie. Cependant, il est possible de préciser une composante de transfert sur laquelle on analyse un gain de rétroaction, comme expliqué plus bas (option balayage de Borel).

ZSTEER

Après incrémentation de la variable η, on passe par la séquence **ZSteer** qui doit au minimum (fourni avec le code) contenir :

```
! ***************************************************************
! ZSTEER : gestion en fin de pas de temps.
! ***************************************************************
  ZPRINT = mod(istep+1,modzprint).eq.0;
  Zprint = zprint .or. mod(istep+1,modzprint).eq.1;
  Zprint = zprint .or. mod(istep+1,modzprint).eq.2;

  Zout = mod(istep,modzout).eq.0;
```

qui ne fait ici que gérer la fréquence de sortie des impressions et des fichier **.data**. Dans la mesure où on a accès aux matrices jacobiennes, à la matrice d'avance A^{spha} et éventullement à leur dérivées par rapport à un paramètres, de nombreux calculs spécifiques peuvent ainsi y être effectués. Deux exemples sont donnés pour le calcul de sensibilité des valeurs porpres à un paramètre (cf stb et seb, voir **lorhcl**, ou celui des exposants de Lyapunov dans l'exemple **Predator**).

Bizarrement, on n'a pas d'exemple de variation distinguée du pas de temps **dt**, ce qui devrait interpeller l'unsecte.

ZSTEP

Une nouvelle séquence[1]. (vide) située avant les incrémentations temporelles permet sur critère numérique de redémarrer un pas de temps, par l'instruction **goto :RedoStep**[2].

Le calcul en double précision.

Mini_ker passe automatiquement en double précision en levant le drapeau **sel double**. Ceci se fait de deux façons :
- à la place de **mod**, on lance **smod double** dans cmz ; on obtient le .exe sous le répertoire : exo/double, où on effectue les opérations courantes.
- on lève le drapeau double dans la selseq.kumac, et on remplace les calculs en simple par les doubles sous exo/exo classiquement après **mod**.

Il est recommandé de passer en double dés qu'on attaque un problème d'optimisation, comme décrit dans la section 6, mais aussi plus généralement pour vérifier la précision numérique obtenue, conjointement avec une diminution de **dt**.

Le code double est obtenu en utilisant l'option **implicit double precision(a-h,o-y)**, et par une macro Mortran transformant **real** en **double precision**.

ATTENTION les valeurs numériques ne changent pas de codage — ils restent dans la précision de la manière dont ils sont écrits (1. rest toujours en simple, 1.D0 toujours en double. Il ne faut donc pas utiliser de chiffres en argument de fonction, sauf pour les fonctions Fortran implicites (sin, cos) qui changnet automatiquement tous les types.

[1] 102

[2] ceci explique pourquoi le balayage de Borel et le calcul de **aspha** ont été déplacés, le label :redoStep : est situé juste après.

5 Outils d'analyse dynamique des systèmes

5.1 Balayage de Borel

On montre dans l'introduction au TEF-ZOOM comment un schéma de Crank-Nicolson donne symboliquement la solution du système transformé de Borel, en identifiant la variable dTime à la variable de Borel τ à un facteur 2 près. Dès lors, il est possible d'étudier numériquement la dépendance en τ de transformée des divers coefficients de matrices réduites du système. Ansi, en un point de la trajectoire, on peut numériquement obtenir la transformée de Borel en effectuant un balayage sur la variable **dTime**.

Il est prévu de sortir simplement cette dépendance en τ pour la matrice réduite à une variable scalaire ff(k). On définit ainsi un gain de rétroaction $g(\tau)$. On explique dans le document de la page web ZOOM (documents.dir/ClimSIre3.ps.gz) comment on définit et interprète cette fonction de gain ou de facteur de rétroaction attaché à une composante de transfert donnée. Un exemple d'analyse de système est donné dans le même document.

Pour les systèmes linéaires - dont le SLT est statique (ou dit aussi autonome), la fonction de τ du gain ne dépend pas du temps. Mais en général, on analyse la fonction $g(\tau;t)$ le long d'un segment de trajectoire, comme dans l'exemple suivant :

```
! -------------------
! Gain de retroaction
! Borel
! -------------------
ZBorel=.true.;
if ZBorel                                         " activation de l'option"
<  istep_B_deb=1000;                          " debut du balayage (increment #"
   istep_B_fin=1200;                          " fin                            "
   istep_B_inc=1;                             " à chaque pas de temps          "
;
   index_ff_gain=7;                       "composante de ff(.) soumise au calcul"
   tau_B_ini=0.2;                       "valeur initiale de la variable de Borel"
   tau_B_mult=sqrt(sqrt(2.));         "facteur multiplicatif de tau du balayage"
   itau_max=100;                                      "nombre de valeurs de tau"
   z_pr/Borel/:tau_B_mult,tau_B_ini*(tau_B_mult)**itau_max;
>;
```

Dans cet exemple, le balayage sera effectué 100 fois à chaque pas de temps, entre la 1000ième et la 1200ième itération sur la trajectoire. Le fichier **tau_Borel.data** contient ces valeurs de τ, et les gains sont dans **gains.data**, une ligne par point de la trajectoire.

Lorsque l'on entre une valeur initiale de τ négative (tau_B_ini=-0.2 ;), le balayage sera effectué sur 201 valeurs, de -0.2,tau_B_mult$*(-0.2),...,0.$, et les valeurs positives symétriques.

Pour obtenir des surfaces $g(t,\tau)$, on peut utiliser le programme Fortran fourni **gains.f** et sa shell de lancement **gains.xqt**, qui met les sorties au format d'un histogramme de PAW, dans lequel on pourra utiliser la kumac : **borels.kumac** générée par le code. Un exemple de résultat est dans le document http ://lmd.jussieu.fr/Zoom/doc/LorenzGains.ps.gz .

autre écriture Dans le cas d'utilisation de symboles, on peut remplacer la donnée par **index_ff_gain=...** de la composante du transfert par l'instruction :

```
ZBorel for: symbole;
```

le symbole est le nom donné à la variable, avec en Grid1D la convention (inode = .).

Autre calcul du gain dynamique de rétroaction

Avec la version 1.01 apparait un autre calcul automatique des pôles et résidus suivant la méthode décrite dans le texte Effluents (réservé à l'unsecte). On trouve ces résultats dans les fichiers **vpgains** qui donne $g(\tau)$ et **vprepons.data** ($\frac{g}{1-g}(\tau)$), voir la subroutine **Boreleig** (version 101e). Le post-traitement est dans le patch **gsrun**, qu'il faut importer dans son exercice (**smod gsrun**), et un exemple d'utilisation dans **predator** ; fichiers de sorties $g(t)$ et répons efficace (t)e dans **vpgaint et vprept.data**. Exemple encore dans http ://lmd.jussieu.fr/Zoom/doc/LorenzGains.ps.gz et l'article refusé par la *Phys. Rev. Lett.*

5.2 Calcul automatique des sensibilités

Une deuxième utilisation directe des matrices Jacobiennes du système permet le calcul automatique de trois types de sensibilités :

- la sensibilité des variables d'état à la perturbation des conditions initiales de l'une d'entre elles.
- la sensibilité des mêmes à un pulse ou un step initial sur un transfert.
- la sensibilité des mêmes à la modification d'un paramètre du modèle.

Ces calculs sont démarrés soit par défaut à $t = 0$, soit à un autre instant. Nous décrivons d'abord les deux premiers, pour lesquels on déclare chaque variable, et le type de perturbation : :

```
  Sensy_to_var
  <
!*     var: eta_courant_L, pert: init at 100;
!*     var: ff_T_czcx,     pert: pulse at 100 every 20;
!*     var: ff_Psi_Tczcx,  pert: step_eff;
!*     var: ff_Psi_Tczcx,  pert: step_Resp at 10 every 100;
! *** premiers tests identiques a lorhcl.ref
    var: ff_courant_L , pert: step_eff;
    var: ff_T_czcx    , pert: step_eff;
    var: ff_Psi_Tczcx , pert: step_eff;
    var: ff_Psi_Tsz   , pert: pulse at 100 every 50;
  >;
>;
```

Les instructions se comprennent d'elles-même ; la différence entre step **eff** et **resp** renvoie aux analogues (non stationnaires) entre respectivement $\dfrac{g}{1-g}$ et $\dfrac{1}{1-g}$. Une vérification est effectuée pour distinguer selon l'applicabilité des modes de perturbation aux états (seul INIT est permis) et aux transferts (tous les autres).

La mise-à-jour **102q** propose de plus une amplitude de step modulable : pour une composante k des transferts à laquelle on applique ce step, il suffit de rentrer

```
step_psi(k)= 3.3d-04; "2xCO2 amplitude"
```

Initialisation Par défaut et sans précision de **at III**, les sensibilités de type état **INIT** sont initialisees à la valeur unité — correspondant à la matrice unité pour un propagateur $\Phi(0, 0$ complet. On peut modifier ces valeurs en remplissant soit-même le tableau **Phi_t** après la fermeture du bloc **sensy_to_var**. Exemple, la cinquième variable d'état est une concentration de CO2 ($\approx 3.3E-04$: une perturbation de 1 mène à des sensibs énormes ; on met alors **Phi_t(5,5)**= $3.3E-04$, et on directement les sensibs à un doublement de CO2.

Dans le cas où on lance ce calcul à un pas de temps non initial (**at 100**), il faut effectuer la mêmeinitialisation de **Phi_t(.,.)** dans **$ZSTEP**, mais aussi anuller la (les) colonnes correspondantes de **dPhi_t**. On aura une séquence comme ça :

```
if istep.eq.100
<      call veczero(dPhi_t(1,5),np); Phi_t(5,5)=3.3E-04;
       print*,' ---  resetting C.I. sensy to State #:',5,' at:'
                                  ,time,istep,' on:',5,'th Phi_t colon.';
>;
```

Sorties Les fichiers de sortie comprennent deux premières lignes de commentaire (gnuplot), et sont :
- pour les états **sens.data** ;
- pour les transfers **sigma.data** ;
- pour les probes **sigmo.data** ;

Si l'option de génération de DimEtaPhi n'est pas retenue, il faut donner une valeur au paramètre **NXP** dans la séquence. Les paramètrfes nyp et nzp restent nuls - ils correspondent à des développements possibles de l'adjoint.

Il est à souligner que cette analyse de sensibilité utilise les dérivées formelles du modèle établies sous le TEF, et fournit des résultats plus rigoureux que par une méthode consistant a comparer des trajectoires correspondant à des accroissements finis des paramètres[1].

sensibilité à une liste de paramètres.

Supposons que nous voulions étudier la sensibilité des variables à un changement initial du paramètre **apar** du modèle de Lotka-Volterra. Il suffit de rentrer dans son Zinit :

```
Free_parameter: apar;
```

pour obtenir dans les fichiers **sensp, sigmap** et **sigmop.data** la sensibilité des variables au cours de la simulation. Il faut aussi donner le **nombre de ces paramètres** dans \$dimetaphi : lp=1 ici, lorsque l'on n'utilise pas la génération automatique de **DimEtaPhi**.

Forme plus complète :

```
Free_parameter: apar, cpar at 100; "[every 10 000]"
```

Il peut en effet s'avérer qu'un modèle subisse une forte transition initial dont on souhaite qu'elle n'intervienne pas dans la senbilité aux paramètres. Dans le cas présenté, on démarre au 100ème pas de temps. Il peut de plus s'avérer qu'une sensibilité paramétrique "explose", d'où la possibilité de la redémarrer (ici tous les 10 000 pas de temps, en commentaire).

remarque étonnante : supposez que dans vos équations, la valeur numérique 3.14159 apparaisse – et si répétée, avec toujours exactement la même liste de caractères ; eh bien, on peut demander : **Free_parameter : apar, 3.14159 ;** *! On aura en sortie les sensibilités à une modification de CE patamètre. Qui trouvera mieux que Mortran ? Remarquons une fois de plus que ça ne marche pas avec le type Fortran 0.31E01 !*

[1] Pour une explication succinte du calcul automatique des sensibilités, voir le document http :lmd.jussieu.fr/zoom/documents.dir/sensibilite.ps. Le document plus complet Propagateurs est réservé à l'unsecte

sensibilités diverses à un paramètre.

Un drapeau **dPi_aspha** dans la selseq.kumac active le code calculant la sensibilité de la matrice d'avance de phase M (**aspha**) par rapport au paramètre désigné comme **[mx :apar],cpar** dans la liste des **free_parameters**. La sortie est dans le tableau Fortran **d_pi_aspha(n,n)**. Le calcul effectué est le suivant :

$$\begin{aligned} d_\varpi M = \quad & d_\varpi H + d_\varphi H \, d_\varpi \varphi \\ + \quad & \{ d_\varphi B \, d_\varpi \varphi + d_\varpi B \} \, (I - D)^{-1} C^\dagger \\ + \quad & B \left\{ (I - D)^{-1} d_\varpi C^\dagger - (I - D)^{-1} d_\varpi D \, (I - D)^{-1} C^\dagger \right\} \end{aligned} \tag{6}$$

On montre comme exemple d'utilisation[1] que la sensibilité de valeurs propres à ce paramètre est donnée par :

$$< f_i \, e_i > d_\varpi \lambda_i = < f_i \mid d_\varpi M \mid e_i > \tag{7}$$

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. Il faut déclarer les tableaux nécessaires dans **ZINIT** et les calculs sont effectués dans le **$ZSTEER** (cf exo **lorhcl**).

[1] Une idée de stb, avec, ici, la version où seul un transfert est explcitement dépendant de ϖ.

5.3 Utilisation du modèle adjoint

De façon générale, le modèle adjoint permet de calculer les gradients d'une fonctionnelle de coût J à des perturbations, le long du SLTC[1] :

$$J = \psi[\eta(T), \varphi(T), h(T)] + \int_0^T l[\eta(\tau), \varphi(T), h(\tau)]\, d\tau \tag{8}$$

Exemple, où on active l'option de calcul de l'adjoint **ZBack** :

```
! ---------------------------------------------------
!   Adjoint  (correspond a l'exemple des masselottes)
! ---------------------------------------------------
Zback=.true.;
! Initialisation v(T) -> pour zinit-adjoint
  Fun_set cout_Psi = eta_move(inode:1);
! et fun_set cout_l integrande de la fontion de cout
  Fun_set cout_l = 0.;
```

Il suffit de donner les deux fonctions correspondant au calcul de la fonctionnelle J : cout_Psi (pour $\psi(\eta(T)$, le coût dit final), et cout_l correspondant à l'intégrande. Dans l'exemple particulier, la fonctionnelle est réduite à la valeur finale d'une composante de l'état. A l'instant $t = 0$, le vecteur adjoint à l'état sera égal à la sensibilité de l'état final de cette deuxième composante (position de la première masselotte) à la perturbation de l'état initial des positions et vitesses de tout ces objets. De façon plus générale, on obtient le gradient de la fonction de coût[2]. Lors du calcul, les variables v_adj(.) et w_adj(.) sont respectivement les adjointes à eta(.) et ff(.). On les récupère dans les fichiers **vadj.data** et **wadj.data**.

sensibilité à des paramètres.

On obtient très simplement en plus la sensibilité de la fonction de coût à des paramètres du modèle dans le fichier : **gradpj.data** en donnant la liste des paramètres dîts alors "libres" dans le Zinit :

```
Free_parameters: apar, cpar ;    " pour predator"
Free_parameters: rd(1), rk(2) ;   " pour masselottes"
```

que l'on déclare après l'initialisation numérique des variables FTN définies comme paramètres libres. (L'option [fwd : .] peut porter en plus sur un terme de la liste. **Il faut aussi donner le nombre de ces paramètres dans $dimetaphi : lp=2 ici.**

système avec loi de commande

Si le problème comprend une loi de commande, on doit activer le drapeau **ZCommand** dans ZINIT. Les deux types de commandes sont données par les variables **ux_com(1 : np)** et **uy_com(1 : mp)** commandant les états et les transferts, composante par composante.

Deux types de commandes sont possibles : a) soit elles sont données comme fonction des variables du système, et le drapeau **ZLaw** doit être levé (cf l'exemple **pousse**), b) soit elles pilotent à chaque pas de temps le système par valeurs numériques dans les fichiers **uxcom** et **uycom** (cf Adjoint et Mini_ker).

La commande par fonction résultera en l'écriture de ces deux derniers fichiers. Pour entrer une loi de commande, lever le drapeau **Zlaw** flag, et entrer la loi dans la séquence

[1]Système Linéaire Tangent Circulant le long d'une trajectoire

[2]Les explications détaillées du fonctionnement de l'adjoint sont données dans le document : http :www.lmd.jussieu.fr/documents/Adjoint.pdf ou .ps.gz

$ZCMD_LAW du patch ZinProc (voir l'exemple **pousse**). Si l'option d'entrée par fichier est choisie, l'utilisateur est censé effectuer lui-même la recherche d'optimisation entre deux simulations. On a provisonnellement prévu un embryon de programme dans le PATCH **ucom**. On pourrait par exemple choisir de piloter le paramètre **apar** dans le modèle de Lotka-Volterra. Dans ce cas, on remplace l'occurence du paramètre dans l'équation d'évolution des proies par la variable réservée **ux_com(1)** et, soit définir une loi dans la séquence $ ZCMD_LAW , soit écrire les valeurs instantannées du pilotage dans le fichier uxcom.data avant calcul dans Mini_ker.

Remarque Il est supposé que chaque composante de commande n'interviend que dans l'équation de même indice. Si cette simplification s'avérait trop restrictive dans l'avenir, le code devra être légèrement modifié. Il est d'ores et déjà possible d'utiliser des composantes de transfert pour le faire (pendant qu'il est chaud).

A l'heure actuelle, n'ont été effectuées que des optimisations avec Minuit. Horws, depuis l'avènement de la **102**, il devient possible d'utiliser la matrice complète de sensibilité à la liste des paramètres devant être dèterminés, ce qui donne accès à la méthode de Newton (cf Cahiers), en utilisant par exmple la subroutine du PSI :**OLLSQ**. Nous attendons un exercice pratique pour décrire en détail cette méthode dand Mini_ker.

6 Optimisation avec l'adjoint et Minuit

*La description suivante reste succinte et le lecteur est renvoyé au manuel Minuit du CERN pour une introduction plus complète. Il est utile aussi de regarder les subroutines citées dans //***pousse/minuik*** *et* ***lorhcl/minuik****, ce dernier effectuant un ajustement des conditions initiales sur une trajectoire de référence d'un Lorenz 63.*

Le CERN a développé un programme très élaboré permettant de maximiser une fonctionnelle calculée dans la fonction générique dénommée FUN : **Minuit**. Minuit n'est pas une boite noire qui entreprendrait de faire un fit. Il s'agit plutôt d'une boite à outil permettant sobrement d'enchainer des algorithmes sans se préoccuper de les programmer, de mettre en forme des sorties, et tout le reste.

On construit son menu d'appels à Minuit à partir d'un programme principal du patch **minuik (DSDQ)**. On a prévu une interface avec Mini_ker qui est la subroutine **FMINI** dont le rôle est de passer les arguments de l'un à l'autre dans les deux sens (voir ces programmes dans "pousse". Il est dirigé par Minuit via le drapeau IFLAG.

```
IFLAG =
 0 : premier run pour inititialiser la commande\index{loi de commande} (ou les paramètres).
 1 : calcul de la valeur FUN à minimiser (-> calcul direct seul).
 2 : demande de Fun et de ses gradients       (-> adjoint aussi).
 3 : calcul final avec meilleur fit. Mini_ker en profite pour
     donner un listing complet, calcule les .kumacs\index{kumac (macro CMZ)} etc.
 4 : Fun seult (jamais de calcul de gradients : pour simplex ou scan).
```

Minuit dirige les appels à Mini_ker en suivant une stratégie, lourde par défaut quand on minimise par MIGRAD avec gradients conjugués, calcul des matrices de covariance des paramètres etc. On a de nombreux enchaînements de IFLAG 4 et 2 suivant l'algorithme : typiquement, on aura facilement npar x 500 appels à la fonction (npar=nbre de paramètres) ! Mais en le pilotant correctement, 50 x npar peuvent suffire.

La fonction FUN est ici le "Principal" de Mini_ker, qui se transforme en subroutine quand on exécute sous cmz la commande

> **smod minuik**

ou

> **mod minuik**

mais avec le sel flag (monitor levé dans la selseq.kumac, cf option Monitor).

L'exemple donné dans "pousse" consiste en un problème de commande optimale, la commande consistant en la force de poussée à déterminer à chaque instant pour minimiser une fonction de coût (cf manuel Adjoint et ...). On sélectionne arbitrairement 11 paramètres seulement parmi les 1001 instants de commande dans le tableau uxcom(2). On demande à Minuit de minimiser - cout_J (cout_J étant à maximiser), et on interpole la commande entre ces valeurs.

FMINI est standard à ceci près qu'il faut extraire du vecteur commande sa seule composante active (ou plusieurs en général). Par contre, **DSDQ** est très dépendant du problème puisqu'il pilote la maximisation. **Attention** : il est impératif avant un fit de définir correctement certains paramètres de Minuit. En particulier, les paramètres **TOL** et **UP** sont essentiels. En gros, UP devrait etre un pourcentage du min de J attendu, et Minuit considère que l'erreur sur les paramètres correspond au passage de $J = J_{min}$ à $J = J_{min}+$ UP. Donc, typiquement, si on travaille en écarts quadratiques avec des incertitudes bien choisies, on aura UP=1., c'est le défaut. TOL (tolérance) est le paramètre critère de convergence : on a convergence dès que la distance estimée au minimum (EDM, Estimated Distance to Minimum) est inférieure à 0.001∗TOL∗UP dans MIGRAD (à moins que l'on ai dépassé MAXCALL). Avec SIMPLEX, on prend simplement comme critère TOL - ou 0.1∗UP par défaut.

La subroutine **TINTER** se trouve dans la Mini_ker file (mathlib). Elle fait l'interpolation entre les (par ex.) 11 commandes fixées par Minuit, et de plus, au point d'entrée **TINTERG**, elle propage les gradients de toute les commandes sur les paramètres sélectionnés pour Minuit (ici, les 11). Ce problème d'interpolation entre les instants significatifs et les instants numériques (fixés par un critère numérique sur le pas de temps) est classique et devra toujours être considéré, faute de sous-estimer fortement la sensibilité de la fonction de coût à la commande.

Stratégie de DSDQ pour cet exemple : Le dernier paramètre joue (comme c'est assez généralement le cas) un rôle particulier : ici, il a peu d'effet direct sur cout_J. On le détermine par un **SCAN(n)**. Minuit calcule n fois cout_J et prend la meilleure valeur de ce paramètre, et produit une très jolie courbe ascii de convergence :

```
MINUIT WARNING IN SCAn
============== VARIABLE  11 IS AT ITS LOWER ALLOWED LIMIT
SCAN OF PARAMETER NO. 11,  Ucom11
                ..........................................................
                 .                          .                            .
 -75000.00      ...                         .                            .
                 .                          .                            .
 -85000.00      ...                         .                            .
                 .                          .                            .
 -95000.00      ...                         .                            .
                 .                          .                            .
 -105000.0      ...                         .                           *.
                 .                          .                    ******* .
 -115000.0      ...                         .              *******       .
                 .                          .       ********             .
 -125000.0      ...                         *********                    .
                 .            *************&                             .
 -135000.0      ... ************            .                            .
 -140000.0      ..........................................................
                 /         /         /         /         /         /
           -2.000     18.00     38.00     58.00     78.00     98.00
                         ONE COLUMN= 2.000000       Overprint character is &

FCN=  -132574.8     FROM SCAn      STATUS=IMPROVED     100 CALLS      101 TOTAL
                    EDM= unknown      STRATEGY= 1      NO ERROR MATRIX

 EXT PARAMETER                CURRENT GUESS      PHYSICAL LIMITS
 NO.   NAME        VALUE           ERROR       NEGATIVE       POSITIVE
  1    UCom1       50.000         1.0000         0.0000         110.00
  2    UCom2       50.000         1.0000         0.0000         110.00
  3    Ucom3       50.000         1.0000         0.0000         110.00
  4    Ucom4       50.000         1.0000         0.0000         110.00
  5    Ucom5       50.000         1.0000         0.0000         110.00
  6    Ucom6       50.000         1.0000         0.0000         110.00
  7    Ucom7       50.000         1.0000         0.0000         110.00
  8    Ucom8       50.000         1.0000         0.0000         110.00
  9    Ucom9       50.000         1.0000         0.0000         110.00
 10    Ucom10      50.000         1.0000         0.0000         110.00
 11    Ucom11     0.10490E-03     1.0000         0.0000         110.00
                               ERR DEF= 0.100E+04
**********
**    6 **FIX   11.00
**********
```

ceci fait, on a fixé ce paramètre qui ne sera plus libre dans Minuit. On essaie ensuite un simplex, puis un migrad (très coûteux). On termine le fit par un calcul des matrices de covariance et d'erreur (MINOS. On finit par deux appels à contour de corrélation entre couples de paramètres pour vérifier la quadrique de convergence :

```
**********
**   12 **CONTOUR   3.000       6.000       2.000       25.00
**********
Y-AXIS: PARAMETER   6: Ucom6
   84.97             2222222
   82.05          2222  *  2222
   79.13        222     *     222
   76.21       22       *       22
   73.29      22        *        22
   70.37     22         *         22
   67.45     2        111111       2
   64.53    22      111 *  11
   61.61    2      11   *   11
   58.69   22     11    *    11
   55.77   2      1     *     11
   52.85   2      1     00     1
   49.93   2******1*****00*****1****
   47.01   2      1     00     1
   44.09   2      1     *     11
   41.17   22     11    *     1
   38.25    2      11   *    11
   35.33    22      111 *  111
   32.41     2        111111       2
   29.49     22         *         22
   26.57      22        *        22
   23.65       22       *       22
   20.73        222     *     222
   17.81          2222  *   222
   14.89             22222222
           I            I          I
        44.01                   110.0
                    80.25
      X-AXIS: PARAMETER  3: Ucom3       ONE COLUMN=   2.640
FUNCTION VALUES: F(I)= -0.1672E+06 +   1000.     *I**2
**********
**   13 **CONTOUR   2.000       7.000       2.000       25.00
**********
Y-AXIS: PARAMETER   7: Ucom7
   75.00   3           22222222
   72.08            2222   *  2222
   69.17          222      *     222
   66.26        222        *       2
   63.35       22          *
   60.44      22           *
   57.53      2         111111
   54.61     22       111  * 111
   51.70     2       11    *   111
   48.79    22      11     *     1
   45.88    2       1      *     11
   42.97    2      11     00      1
   40.06   *2******1******00******1*
   37.15    2      11     00      1
   34.23    2       1      *     11
   31.32    22      11     *     1
   28.41     2       11    *   111
   25.50     22       111  *  11
   22.59      2         1111111
   19.68      22           *
   16.76       22          *
   13.85        222        *       2
   10.94          222      *      22
   8.029            2222   *  22222
   5.117   3           22222222
           I              I       I
        50.62                   110.0
                    89.05
      X-AXIS: PARAMETER  2: UCom2       ONE COLUMN=   2.375
FUNCTION VALUES: F(I)= -0.1672E+06 +   1000.     *I**2
```

Les quadriques sont interrompues par la borne maximale donnée aux paramètres 2 et 3.

STRATEGIE de MINUIT

Sans gradients fournis, il en effectue le calcul par différences finies, soit 4 appels par paramètre ($\pm$ delta puis $\pm$ $.1\times$ Delta, pour vérifier la linéarité). En mode gradient, il calcule la direction de minimisation, mais effectue les appels à la fonction en modifiant successivement chaque paramètre. On peut utilement consulter res.lis de pousse.

S'il n'arrive pas à la précision demandée, il entreprend un autre essai en fixant successivement chacun des paramètres libres et en recalculant un minimum avec les autres (recherche de pathologies).

On peut lui en demander moins en donnant la valeur 0 à STRATEGY (défaut=1, max=2).

Remarque sur la précision : On est en général déçu par la précision obtenue. Les calculs dans Minuit sont faits en double précision, et on pousse l'utilisateur à utiliser l'option **double** ou une machine de 64 bits pour Mini_ker.

TOURS DE MAIN

Une fois lancée l'exécution, on peut à tout moment faire – soit un **tail** sur uxcom.data pour suivre l'évolution du fit – soit plus complètement :

> **Crtl z** pour interrompre et regarder les divers fichiers on remet en

> **bg** c'est-à-dire en background pour relancer.

Pour interrompre de nouveau, mettre en foreground puis interrompre :

> **fg**

> **Ctrl z**

Minuik demande à Principal une sortie complète des zprint tous les 100 appels par Minuit (trois sorties successives).

Il est très pratique de "tracer" la progression de Minuit en suivant les exemples donnés (subroutine FMINI) qui génèrent un fichier **Minuik_trace.data** (cf exos pousse et lorhcl).

7 Le filtre de Kalman

version encore non raccrochée à la 102. Le vecteur des obs est à présent donné par les set_probe :¡¿, mais à part ça les principes sont les mêmes. On a cependant grandement amélioré les résultats avec un nouvel algorithme d'avance de la matrice de var-covar. Attention, dans ce paragraphe, μ désogne un bruit alors que ω est le vecteur des observables (noté μ ailleurs.

[1]Le filtre de Kalman permet d'assimiler des données en cours de simulation. On suppose ainsi l'existence d'un modèle réel avec perturbation stochastique de ses états, et qu'on dispose d'observations bruitées de sa réalité. La situation prise en compte dans Mini_ker correspond à des états continument perturbés et à des obervations échantillonnées. On a donc un système sous TEF du type :

$$\begin{aligned} \partial_t \eta(t) &= g(\eta(t), \varphi(t)) + W(t)\mu && (9) \\ \varphi(t) &= f(\eta(t), \varphi(t)) && (10) \\ \omega(t) &= h(\eta(t), \varphi(t)) + \nu && (11) \end{aligned}$$

Les obsevations ω sont disponibles aux temps discrets $t = s_i$. La perturbation stochastique μ est caractérisée par sa matrice de variance-covariance Q, et ν par R. W effectue l'impact sur η des perturbations. A chaque instant d'observation, le filtre met à jour la meilleure estimation des états et de leur matrice de vcov au sens où l'erreur résiduelle est orthogonale à l'accroissement d'information déduite des obs bruitées.

L'exemple suivant est linéaire et perturbé-bruité. On a trois états, trois transferts-copie, mais on a deux sources de perturbations qui agissent sur les trois états.

$$\left\{ \begin{array}{ll} \partial_t \eta_1 = & a_{11}\eta_1 + a_{12}\varphi_2 + a_{13}\varphi_3 + W_{11}\mu_1 + W_{12}\mu_2 \\ \partial_t \eta_2 = & a_{21}\varphi_1 + a_{22}\eta_2 + a_{23}\varphi_3 + W_{21}\mu_1 + W_{22}\mu_2 \\ \partial_t \eta_3 = & a_{31}\varphi_1 + a_{32}\varphi_2 + a_{33}\eta_3 + W_{31}\mu_1 + W_{32}\mu_2 \end{array} \right. \qquad (12)$$

$$\left\{ \begin{array}{ll} \omega_1 = & \varphi_1 + \nu_1 \\ \omega_2 = & \eta_2 + \nu_2 \\ \omega_3 = & \eta_3 + \nu_3 \end{array} \right. \qquad (13)$$

Codage du filtre de Kalman

Il faut lever le drapeau **sel kalman** dans la selseq, et la variable logique Zkalman=.true. dans le zinit. Il faut aussi donner la dimention de la source de perturbation ainsi que celle du vecteur des observations dans **dimetaphi** :

```
parameter (nobsp=3,nerrp=2);
```

On a ici trois obs et deux sources de perturbation.

L'observation et les perturbations

Dans **zinit**, on doit définir un "modèle diagnostic d'observation", qui est un ensemble de fonctions de η et φ qui devra être ajusté aux vraies obs provenant de l'extérieur du run (censé interroger le modèle réel). Exemple :

```
set_probe
< eqn: Obs_tef(1) = ff(1) ;
  eqn: Obs_tef(2) = eta(2);
  eqn: Obs_tef(3) = eta(3);
>;
```

[1]traduite librement du texte anglais de pat

Le nom **Obs_tef(.)** est bine sûr arbitraire. La vcov du bruit est constante et entrée dans le tableau **covobs(mobs,mobs)** :

```
covobs(1,1) = 0.3; covobs(1,2) = 0.;  covobs(1,3) = 0.;
covobs(2,1) = 0.;  covobs(2,2) = 0.1; covobs(2,3) = 0.;
covobs(3,1) = 0.;  covobs(3,2) = 0.;  covobs(3,3) = 0.2;
```

En cours de simulation, il est nécesaire d'indiquer au moteur de l'assimilation que des mesures sont disponibles. Cela se fait dans **zsteer**, en levant le drapeau ZGetObs et en remplissant le vecteur **vobs(mobsp)**, qui sera ainsi pris en compte dans le traitement du pas de temps.

Enfin, la vcov des états doit être donnée dans le tableau **coveta(np,np)** :

```
coveta(1,1) = 0.2;   coveta(1,2) = 0.001;
coveta(2,1) = 0.001; coveta(2,2) = 0.1;
```

avec en plus la matrice d'impact des perturbations sur les états (W) dans le tableau **mereta(np,nerrp)** :

```
mereta(1,1) = 1.;   mereta(1,2) = 0.;
mereta(2,1) = 0.;   mereta(2,2) = 1.;
mereta(3,1) = 0.5;  mereta(3,2) = 0.5;
```

Initialisation

Enfin, on doit initialiser la matcov des états qui sera ensuite calculée par le filtre : tableau **covfor(np,np)** :

```
covfor(1,1) = 1000.; covfor(1,2) = 10.;   covfor(1,3) = 10.;
covfor(2,1) = 10.;   covfor(2,2) = 5000.; covfor(2,3) = 5.;
covfor(3,1) = 10.;   covfor(3,2) = 5.;    covfor(3,3) = 2000.;
```

Résultats

Les résultats de la simulation ajustée sont dans les fichiers standard (**res, tr, dres.data**), on a en plus un fichier **obs.data** rassemblant les observable calculées par le modèle ayant servi à l'ajustement.

8 L'option Monitor

Il est possible de piloter les simulations à partir d'un programme principal appelant Mini_ker (cette option transforme donc comme pour **Minuik** le "principal" en subroutine). Dans cas, l'appel se fait avec les arguments impératifs :

```
call principal(Cost,ncall,IFlag,osuffix,info,idxerror);
```

L'arguments **osuffix** sert à distinguer les fichiers de sorties : il remplace le **.data** standard. Un tableau des principales causes de déroutement de Mini_ker est donné par les arguments de sortie info et idxerror :

La structure choisie est la suivante :
- a) on définit un **patch** de nom quelconque (machin), dans lequel on code son programme pilote ;
- b) On lève le flag **sel monitor** dans la selseq.kumac ;
- c) On assemble et éxécute par **smod machin**. Les sorties sont alors dans le répertoire machin, sous lequel on a un cfs etc

Source of error / warning	index	idxerror
state matrix inversion in ker	LaPack	1
system resolution error in ker	LaPack	2
state matrix advance in phase space,	principal	3
analysis, state matrix advance (aspha), (I-D) inversion	LaPack	21
analysis, Choleski factorization, non pos-def var-covariance mx	—	22
analysis, error matrix inversion	LaPack	23
variance covariance update system resolution (Kalman)	—	24
linearity problem for transfers	principal	-1
Newton D_loop does not converge	newt	-2

TAB. 1 – Monitoring errors and warning from Mini_ker

Les divers paramètres en plus des arguments nécessaires au pilotage de Mini_ker seront rangés dans un **common/interface/** . On devrait aussi convenir de noms spècifiques pour la liste des paramètres (genre **pi(.)**).

Comme exemple d'utilisation, Pat pensait à une étude des bifurcations par variation paramétrique. Mais de façon générale, pour l'utilisation de Minuit, on a intérêt a utiliser cette option. Donnons l'exemple de StepH qui cherche à définir les paramètres d'un modèle macro-éco ayant propention certaine a devenir chaotique, voire singulier pour certaines valeurs des paramètres. Par principe, Minuit explore au maximum le domaine de définition donné et fait ainsi régulièrement tomber le modèle dans des régions singulières. StepH nous a ainsi expliqué au Zircon "Plateforme" qu'il était indispensable de gérer les erreurs pour empècher Mini_ker de se vautrer. Les erreurs détectées (cf tableau 1) aiguillent vers **ZSteer**, où l'utilisateur renvoie au programme père. Dans le cas cité, il faut "signaler" à Minuit que la région explorée est mauvaise, ce que StepH fait en renvoyant un coût de 10^9 et des gradients de 10^{-3}.

9 L'option ZObs

ZGetObs Voir Pat et STB, j'ai pas tout suivi l'utilisation de cette possibilité, mais il semble s'agir d'un point de rencontre (dans $ZSteer) de la simulation du modèle avec des instants où des donnèes externes (mesure, référence) sont disponibles. Par exemple, la fonction de coût (cout_J) n'accumulerait des écarts quads que pour ces instants. Il faut signaler au programme par **Zobs=.TRue. ;** qu'il existe des données, et synchroniser, avec la variable logique **ZGetObs**, le tempo de la mesure avec celui de la simulation.

10 Autres outils d'analyse

Analyse de stabilité des modes rapides : option SLT

Les analyses précédentes sont réalisées en même temps qu'un parcours de trajectoire. On peut également analyser de manière plus classique la dynamique du **modèle d'état** grâce à la matrice A_{st}. La macro cmz **smod SLTC** permet de préparer un programme principal (**circul** dans le +PATCH SLTC), fourni en standard mais prévu pour être modifié au gré de l'utilisateur. En standard, on calcule une SVD (Décomposition en Valeurs Singulières) à la fois de A_{st} et $(A_{st} + A_{st}^{\dagger})$. Les éléments singuliers de ces deux matrices correspondent à l'analyse des modes de croissance les plus rapides du système perturbé. Voir directement le programme */sltc/circul* bien commenté.

Attention, avant de lancer un run sous ./..exo../sltc/, il est impératif de créer deux liens, sur deux fichiers de /..exo../. Sur **title.tex** pour transmettre un titre au graphique, et **aspha.data** pour l'accès à la matrice d'avance d'état.

Sorties : **u,w,v.data** et **us,ws,vs.data** pour les éléments singuliers des deux matrices. leurs macros .kumac correspondantes pour PAW)[1].

Analyse du propagateur sur le SLTC

La matrice A_{st} est utlisée pour calculer le propagateur sur le SLTC. Le calcul est effectué par produits finis du développemnt au cinquième ordre de $\Phi(t+\delta t, t) = \exp A_{st}\delta t$. Divers éléments d'analyse sont fournis en standard, dont en particulier les exposants de Lyapunov (cf programme **circule** dans le patch SLTCIRC. On prépare une exécution en lançant la macro smod SLTCIRC sous cmz. Exécutable et sorties (avec leurs kumacs) dans /..exo../sltc :

```
              ntuple # var name
--------------------------------------------------------------------
phit.data       55      p[ij]
uphit.data      50      up       SVD de Phit
wphit.data      51      wp        = U[w_diag]V'
vphit.data      52      vp
wr.data         53      wr[i]    éléments propres
wi.data         54      wi       partie réelle et imaginaire
--------------------------------------------------------------------
lphit.data      65      lp[ij]   1/t Ln Phit     + 1/LnDetPhit,<TrA>
ulphit.data     60      ulp      SVD de 1/t LnPhit
wlphit.data     61      wlp
vlphit.data     62      vlp
wrl.data        63      wrl[i]   VP de 1/t LnPhit
wil.data        64      wil
--------------------------------------------------------------------
lwphit.data     67      lwp[i]   1/t Ln w : Phit=UwV' (Lyapunov)
lwr.data        68      lwr      1/t VP de UV'
lwi.data        69      lwi
--------------------------------------------------------------------
```

Le bloc 60-65 contient les éléments d'analyse du propagateur assimilé à un système autonome $\Phi(t, 0) = \exp At$ où A serait la matrice d'état autonome d'un système donnant le même propagateur au temps t. Le dernier bloc 67-69 donne les exposants de Lyapunov ordonnés du système et les éléments propres associés.

Trois liens doivent être créés avant de lancer une exécution, les deux du SLTC, avec un troisième sur **dres.data**, ceci pour avoir accès à l'incément de temps du calcul le long de la trajectoire[2].

Analyses en ligne

Comme on l'a déjà remarqué, l'utilisateur a toute laltitude pour extraire les quelques lignes des nombreux calculs de SLT et SLTC qui l'intéresse et les programmer dans la séquence **ZSteer** en dimentionnant les tableaux nécessaires dans le **ZINIT**. Toujours penser aux règles pour passer sans problème en double précision (en particulier le nom générique des fonctions FTN sin, cos, min, etc).

---

[1]Explications dans le papier sur le SLTC (Al1 2003)

[2]Pour la recherche sur l'analyse du propagateur, consulter les documents sur le SLTC et les Gains sur champs (Al1 2003-2004)

Annexe sur la programmation de Mini_ker

On donne ici des indications sur la manière dont les macros, et en particulier celles de la dérivation partielle, arrivent à faire le travail, sans effets magiques.

Dans cette boucle extraite de **principal**, on calcule la matrice jacobienne $Bb(,j) = \partial_{\varphi(j)}\eta(i)$, ligne par ligne, pour la partie maillée :

```
     do j=1,mp
     < do inode=1, n_node
       < ;mult[i]=1,n_mult : Bb(k2i_[.i](inode),j)
                           = F_D(nde_eta_fun_[.i](inode))(/ff(j));
       >
     >;
;
! Build TEF B mx from Bd
         call scamat(-dt/2.,Bb,B,n,n,m,m);
```

la macro **mult** génère n_{mult} lignes de code calculant chacun un élément de la matrice. La macro **F_D(fun)(/ff(j)** va effectuer la dérivation symbolique de la fonction **fun** par rapport à la variable ff. Le nom générique de la fonction associée à la ième variable d'état d'une maille (nde_eta_fun) est associée a celle définie par le bloc Mortran set_node_eta<> du zinit. Ainsi, à la précompilation par Mortran de ces instruction, la partie fonction est remplacée strictement par l'expression donnée dans le ZINIT, fonction de η et φ ou de leurs symboles (var :) donnés dans le même bloc de ZINIT. L'appel à **scamat** est réalisé pour passer à la matrice B du système discrétisé en temps du TEF (multiplication par $\delta t/2$).

Fortran généré (cfs/principal.f) :

```
10125 DO 10129 j=1,mp
      DO 10131 inode=1,n_node
      Bb(k2i_1(inode),j)=(0.)
      Bb(k2i_2(inode),j)=( (((-F_delta(k2j_1(inode+1),j))+F_delta(k2j_1(
     *inode),j))+((-F_delta(k2j_2(inode+1),j))+F_delta(k2j_2(inode),j)))
     **(rmassm1(inode)))
10131 CONTINUE
10132 CONTINUE
10129 CONTINUE
```

On reconnait les fonctions dérivées de deta_move et deta_speed du ZINIT des masselottes. La dérivation est faite sans prise en compte de l'indice de la variable ff, et la dérivée est multipliée par la fonction déclarative (Fortran "statement function") **F_Delta(k,l)** qui vaut un pour $k = l$, zéro sinon.

Structure des répertoire, installation et lancement Mini_ker en résumé

Partie reprise d'un **Help** *ancien et peut-être redondante.*

`La structure informatique de mini_ker est la suivante :`

```
.----------------------------------------------.
|  .../mini_ker/                                |
|                                               |
|   cmz file contiend la partie commune         |
|   aux exos. En mode exo, elle reste inchangée.|
`----------------------------------------------'
               dessous :
         /------------------------\
         |        |        |       |
       /exo1    /exo2  [/lorhcl /pousse]
```

pour chacun : une cmz file avec :
- le patch zinproc qui contient toute l'information du modèle [calcul direct trajectoire, sensibilités, adjoint
- patch sltc : SVD de la matrice d'évolution des états (Singular Value Decomposition).
- patch sltcirc : calcul du propagateur et de ses éléments singuliers.
- command : gestion de recherche d'une commande optimale hors Minuit. [ce dernier est a l'état embryon correspondant à l'exo pousse]
- minuik : recherche de maximum de fonctionnelle par **Minuit**

Démarrage après réception

Avant de lancer cmz, on édite **cmzlogon** pour spécifier un éditeur :
- par défaut à Jussieu, c'est **ce cv** (Apollo like)
- en décommentant les deux commandes xedit, on invoque cet éditeur

attention, les options qui sont éditeur-dépendantes sont capitales, il faut que cmz reste en attente lors de l'édition d'un fichier.

on lance cmz (les helps sont en ligne). se familiariser avec un des exemples fournis : pour éxécution de calcul trajectoire : **exe mod** (ou mod)

crée **exo/exo, exo/exo/cfs**

on lance l'exécution dans exo/exo/. [exo.exe >res.lis]

on obtient un ensemble de fichiers file.data avec des kumacs associées, qui donnent le contenu des fichiers file.

Si on utilise **PAW** comme grapheur, il suffit de lancer une kumac (ex : **exe eta** [ou eta], liste des kumacs par macro/list) pour obtenir les trajectoires. *la mémoire sature pour les grandes trajectoires, il peut être nécessaire de faire >hi/del * entre deux kumac.* Un message donne un exemple pour sortir un ntuple, genre :

```
 > NT/PL 51.wp1%t !  1000    1 ! L !   [ou >help nt/pl]
```

on peut faire au max du 4D :

```
 > NT/PL 51.x%y%z%t !  1000    1 ! * !
```

pour obtenir un portrait de phase 3D avec le t → couleur. Et (rappel) pour avoir une palette arc-en-ciel :

```
>set ncol 32
>palette 1
```

Applications persos

Faire son exo :(attention, le tout prend au moins 1/4h avant de sortir les résultats...) On commence par copier une cmz ressemblante, changer le nom et modifier la **cmzlogon.kumac** (ou en passant par un .car, interroger les cmz fans).

dans le patch zinproc, on modifie les séquences pour :

1. donner la dimention de eta phi $DIMETAPHI (*)
2. rentrer son modèle par les fun_set ou équivalentes de $ZINIT etc on a aussi à choisir des flags logiques :
   - **Zback** si on veut un aller-retour avec adjoint et dans ce cas,
     - **Zcommand** si on gère une loi de commande
     - **Zlaw** si on explicite une loi (les valeurs sont alors inscrites dans les fichiers uxcom et uycom. Sinon, on lit ces fichiers en entrée ; ils sont donc censés avoir été remplis par une procédure de recherche d'optimum (cf patch command).
   - Rq $ZSTEER comme dans ZOOM, se situe en fin de boucle (gestion du pas de temps par exemple ou des sorties / modzprint)
3. le lancer comme auparavant.

remarque sur le fonctionnement : avec **mod***, on éxécute la kumac selseq.kumac qui précise qu'on va chercher les sources de ../ qui est le noyau de calcul du Mini_ker .*

Sorties diverses

Dans **exo/exo/cfs**, on a les codes Mortran et Fortran générés.

- le fichier exo_o.tmp est le plus explicite, il contient le programme "principal" en mtn. **principal.f** est aussi a lire pour i) vérifier la valeur donnée aux paramètres de dimentionnement losque l'on en sollicite le calcul automatique ; ii) vérifier le calcul symbolique des dérivées et opérateur du TEF. Les deux sont utiles pour qui veut rapidement se familiariser avec le Mini_ker. Les symboles utilisés sont très proches du papier "Adjoint".

Comme toujours, au lieu de se précipiter sur le graphique, on a intérêt à consulter un peu les sorties **res.lis**. Et en particulier voir si on à un message de warning sur ffl (phi). Un test est en effet effectué à chaque incrément pour vérifier que **ffl(time-dt)+dPhi** reste proche de **ffl(eta(time))**. Sinon, ça veut dire que pb de linéarité (diminuer alors dt). **Remarque :** Ce dernier critère est nécessaire, malheureusement pas suffisant, cala dépend de la manière dont les non-linéarités sont distribuées entre cellules et transferts. Il reste donc conseillé d'effectuer le test de comparaison des résultats par variation du pas de temps avec passage en double précision.

Validation numérique des algorithmes

La validation des méthodes de calcul de la simulation a été faite en comparant le modèle de Lorenz 63 en simulation effectuée avec ZOOM sur station Apollo et Mini_ker sous Linux. On trouve six décimales identiques au bout de 3000 pas de temps (time=s) avant que les résultats ne divergent. Cela constitue une excellente validation, restituant en particulier le même attracteur que celui abondamment décrit dans la littérature.

Il est également possible de mener une intéressante comparaison entre les trois manières de calculer une sensibilité dans Mini_ker :

- avec la matrice A^{spha} et le calcul du propagateur sur le SLTC, on a la matrice de sensibilité des états (t) aux valeurs initiales ;
- l'option Zsensib[1] permet de calculer cette matrice colonne par colonne ;
- et ZBack, avec une fonction finale de cout restreinte à un seul état donne une ligne du propagateur.

Voici ces trois matrices (transposées) pour un autre exercice avec option **Grid1D** :

```
===========================================================================================================
          Comparaisons entre trois sensibilites / Masselottes Thu Oct 21 18:11:15 CEST 2004 All
===========================================================================================================
sens.data (5000 calculs)
1 0.10000E+01  0.00000E+00  0.13165E+01 -0.42256E-01  0.15780E+01 -0.69412E-01  0.17284E+01 -0.79530E-01
2 0.00000E+00  0.10000E+01  0.13165E+00 -0.42258E-02  0.15780E+00 -0.69412E-02  0.17284E+00 -0.79530E-02
3 0.00000E+00  0.00000E+00 -0.55054E-01  0.15101E-01 -0.11106E+00  0.17038E-01 -0.15040E+00  0.10118E-01
4 0.00000E+00  0.00000E+00 -0.42257E-01 -0.53544E-01 -0.69413E-01 -0.10936E+00 -0.79531E-01 -0.14939E+00
5 0.00000E+00  0.00000E+00 -0.11106E+00  0.17038E-01 -0.20546E+00  0.25219E-01 -0.26147E+00  0.27156E-01
6 0.00000E+00  0.00000E+00 -0.69413E-01 -0.10936E+00 -0.12179E+00 -0.20294E+00 -0.14894E+00 -0.25875E+00
7 0.00000E+00  0.00000E+00 -0.15041E+00  0.10119E-01 -0.26147E+00  0.27156E-01 -0.31652E+00  0.42257E-01
8 0.00000E+00  0.00000E+00 -0.79531E-01 -0.14939E+00 -0.14894E+00 -0.25875E+00 -0.19120E+00 -0.31229E+00
vadj (5000 calculs)
  1*           2*           3*           4*           5              6*           7              8*
  0.10000E+00  0.00000E+00  0.13166E+01 -0.42131E-01  0.15788E+01 -0.69155E-01  0.17294E+01 -0.79180E-01
  0.00000E+01  0.10000E+01  0.13166E+00 -0.42131E-02  0.15788E+00 -0.69156E-02  0.17294E+00 -0.79181E-02
  0.00000E+00  0.00000E+00 -0.55089E-01  0.15106E-01 -0.11127E+00  0.17001E-01 -0.15053E+00  0.10024E-01
  0.00000E+00  0.00000E+00 -0.42131E-01 -0.53579E-01 -0.68061E-01 -0.10940E+00 -0.77685E-01 -0.14943E+00
  0.00000E+00  0.00000E+00 -0.11110E+00  0.17001E-01 -0.20577E+00  0.25130E-01 -0.26180E+00  0.27025E-01
  0.00000E+00  0.00000E+00 -0.69155E-01 -0.10940E+00 -0.11928E+00 -0.20301E+00 -0.14575E+00 -0.25883E+00
  0.00000E+00  0.00000E+00 -0.15043E+00  0.10024E-01 -0.26180E+00  0.27025E-01 -0.31704E+00  0.42131E-01
  0.00000E+00  0.00000E+00 -0.79179E-01 -0.14943E+00 -0.14575E+00 -0.25883E+00 -0.18734E+00 -0.31241E+00

 phit (5000 calculs) sltcirc
  0.10000E+01  0.00000E+00  0.13165E+01 -0.42254E-01  0.15780E+01 -0.69421E-01  0.17284E+01 -0.79551E-01
  0.00000E+00  0.10000E+01  0.13165E+00 -0.42249E-02  0.15780E+00 -0.69421E-02  0.17284E+00 -0.79554E-02
  0.00000E+00  0.00000E+00 -0.55048E-01  0.15090E-01 -0.11106E+00  0.17035E-01 -0.15040E+00  0.10131E-01
  0.00000E+00  0.00000E+00 -0.42264E-01 -0.53540E-01 -0.69431E-01 -0.10936E+00 -0.79559E-01 -0.14941E+00
  0.00000E+00  0.00000E+00 -0.11106E+00  0.17035E-01 -0.20546E+00  0.25222E-01 -0.26146E+00  0.27166E-01
  0.00000E+00  0.00000E+00 -0.69419E-01 -0.10936E+00 -0.12180E+00 -0.20293E+00 -0.14897E+00 -0.25875E+00
  0.00000E+00  0.00000E+00 -0.15041E+00  0.10130E-01 -0.26146E+00  0.27166E-01 -0.31651E+00  0.42255E-01
  0.00000E+00  0.00000E+00 -0.79553E-01 -0.14939E+00 -0.14897E+00 -0.25875E+00 -0.19123E+00 -0.31229E+00

 calcul de Phi (avec Psi) en avant version 1.01 (Juin 2005)
  0.10000E+01  0.00000E+00  0.13165E+01 -0.42258E-01  0.15780E+01 -0.69416E-01  0.17284E+01 -0.79535E-01
  0.00000E+00  0.10000E+01  0.13165E+00 -0.42259E-02  0.15780E+00 -0.69416E-02  0.17284E+00 -0.79536E-02
  0.00000E+00  0.00000E+00 -0.55054E-01  0.15101E-01 -0.11106E+00  0.17039E-01 -0.15040E+00  0.10119E-01
  0.00000E+00  0.00000E+00 -0.42259E-01 -0.53544E-01 -0.69417E-01 -0.10936E+00 -0.79536E-01 -0.14939E+00
  0.00000E+00  0.00000E+00 -0.11106E+00  0.17039E-01 -0.20546E+00  0.25220E-01 -0.26147E+00  0.27158E-01
  0.00000E+00  0.00000E+00 -0.69417E-01 -0.10936E+00 -0.12179E+00 -0.20294E+00 -0.14895E+00 -0.25875E+00
  0.00000E+00  0.00000E+00 -0.15040E+00  0.10119E-01 -0.26147E+00  0.27158E-01 -0.31652E+00  0.42259E-01
  0.00000E+00  0.00000E+00 -0.79536E-01 -0.14939E+00 -0.14895E+00 -0.25875E+00 -0.19121E+00 -0.31229E+00
 idem, mais avec dt doublé, cad 2500 calculs
  0.10000E+01  0.00000E+00  0.13165E+01 -0.42263E-01  0.15780E+01 -0.69399E-01  0.17284E+01 -0.79485E-01
  0.00000E+00  0.10000E+01  0.13165E+00 -0.42263E-02  0.15780E+00 -0.69397E-02  0.17284E+00 -0.79486E-02
  0.00000E+00  0.00000E+00 -0.55067E-01  0.15128E-01 -0.11107E+00  0.17047E-01 -0.15040E+00  0.10087E-01
  0.00000E+00  0.00000E+00 -0.42263E-01 -0.53555E-01 -0.69398E-01 -0.10937E+00 -0.79485E-01 -0.14939E+00
  0.00000E+00  0.00000E+00 -0.11107E+00  0.17047E-01 -0.20547E+00  0.25215E-01 -0.26147E+00  0.27135E-01
  0.00000E+00  0.00000E+00 -0.69398E-01 -0.10937E+00 -0.12175E+00 -0.20294E+00 -0.14888E+00 -0.25876E+00
  0.00000E+00  0.00000E+00 -0.15040E+00  0.10088E-01 -0.26147E+00  0.27135E-01 -0.31654E+00  0.42263E-01
  0.00000E+00  0.00000E+00 -0.79486E-01 -0.14939E+00 -0.14888E+00 -0.25876E+00 -0.19115E+00 -0.31231E+00
```

[1] aujourd'hui obsolete.

==

les colonnes de vadj† sans signe $*$ correspondent à 4999 calculs, pour calibrer les comparaisons.

Ces mêmes comparaisons ont été effectuées sur deux autres modèles avec le même score. On remarque en général que le calcul de sensibilité automatique s'approche le mieux du propagateur - que l'on peut prendre comme référence du fait de l'ordre élevé des dévelopements du calcul. Cela se conçoit si l'on remarque que les calculs avec Zsensib sont effectués en prenant en compte les dérivées des matrices (par différences finies).

On a aussi comparé ces trois matrices pour l'exemple macro–éco de stepH (dénommé gras_double par Al1), exercice non-linéaire avec matrice D[1].

D'autres validations numériques concernant les différents modes de perturbation des transferts et le tableau $\Psi(t,0)$ est disponible sur demande. On a pu noter en particulier que le passage en double précision ne fait gagner en gros que deux ordres de grandeur, et que l'on bute manifestement sur une complexité algorithmique.

---

[1] cf document Propagateurs.

Index

Table des matières