Vers la Mini-ker-200

Al1, Octobre 2007

Version 2.xx

Ce cahier accompagne les efforts pour passer de la 102 à la 200.

On y conserve les intermédiaires historiques pour être à même de considérer différentes solutions, et ainsi revenir en arrière si nécessaire. L'horizon de ce texte est de fournir les éléments des Cahiers de progammation de la 200.

Principes et structure du Mini_ker 2.00

On rapelle juste ici les principes déjà énoncés.

Introduction générale aux choix de programmation.
- définition de langages dédiés dès qu'ils se dégagent
- principe d'unicité (ou de non-duplication) des calculs ;

L'écriture de langages pour tenter une remontée de la couche algorithmique est une méthode systématique de la collaboration ZOOM. L'utilisation du précompilateur Mortran[1] offre les facilités nécessaires à l'écriture symbolique des descriptions de modèles et des diverses options de calcul. Remarquons cependant que par rapport à ZOOM, cette systématique reste à un niveau modeste dans Mini_ker-2, du fait qu'un seul niveau d'emboitement est considéré. Les macros de la 200 ont à gérer la création d'un indexage par numéro de familles, et d'un éclatement des quatre matrices de base du TEF en autant de matrices internes aux familles, avec en plus un tranfert externe pouvant coupler tous les objets. Enfin, on "mélange" les précédentes options Grid1D ou non, en imposant qu'une famille interne soit ou non maillée, mais par contre la famille englobante U et son transfert associé[2] suggère par principe de souplesse que l'on puisse alterner des blocs de création de composantes de transfert externes *ad-libidum*. On pourra ainsi alterner des blocs maillés ou non, Up et Dwn à l'envi, avec cette seule restriction que les blocs maillés aient un nombre unique de mailles[3]. Cette dernière facilité permettra en particulier de modéliser un problème 2D sans avoir à créer d'énormes matrices creuses, ce qui aurait été le cas dans la 102.

Le code Fortran généré utilise des séquences de calcul répétées dans des contextes différents, pour le calcul de la trajectoire, des sensibilités et de l'adjoint. On a donc découpé ce code final en séquences (des paquets d'instructions Mortran ou Fortran) ; ces séquences sont gérées par CMZ[4]

[1] SLAC, CERN et RAMSÈS.
[2] dît ici externe, puisqu'il n'y a pas d'ambiguité avec ce seul niveau d'emboîtement.
[3] ceci pour ne pas trop alourdir le code.
[4] CERN Code Management using Zebra.

1 Position du problème l'origine de la 200

Le plus explicite est d'en montrer un exemple de $ZINIT étendant le LORHCL :

```
Under Universe                "====== Famille englobante ======"
< UNDER Part-1                  "------ Famille interne ------"
  < set_eta
    < var: eta_courant_L,
      fun: deta_conv = pi_prandtl*ff_T_czcx
                                      - pi_prandtl*eta_courant_L;
      var: eta_T_czcx,
      fun: deta_energy_zx = - ff_Psi_Tsz + pi_Rayleigh_ratio*ff_courant_L
                                                      - eta_T_czcx;
    >;
    set_Phi
    < eqn: ff_courant_L  = eta_courant_L;       "x"
      eqn: ff_T_czcx     = eta_T_czcx;          "y"
    >;
  >;"end Fm"
;
 UNDER Part-2                   "------ Famille interne ------"
 < set_eta
   < var: eta_T_sz,
     fun: deta_energy_z = ff_Psi_Tczcx - pi_wave_nb_ratio*eta_T_sz;
   >;
   IN_Phi
   < eqn: ff_Psi_Tczcx  = eta_T_czcx*eta_courant_L;    "xy"
     eqn: ff_Psi_Tsz    = eta_T_sz*eta_courant_L;      "zx"
   >;
!-----------------------
! IX-Transfer definition (explicit)
!-----------------------
  IX_Phi
  < eqn: ff_bid_X =  2.*eta_courant_L + 0.*eta_T_sz;
  >;
 >;"end Fm"
!----------------------
! Cell definition
!----------------------
   set_Eta
   < eqn: eta_bid=45.;
   >;
!-----------------------
! IX-Transfer definition (implicit)
!-----------------------
  set_Phi
  < eqn: ff_bid =  eta_courant_L + eta_T_sz;
  >;
>;"End Universe"
```

On a ainsi deux familles internes (part-1 et part-2) sous la famille Universe. On définit dans chauqe famille interne une seule cellule et un seul transfert interne (IN). La famille englobante comporte éventuellement aussi une cellule, et tous les transferts dits externes,

qui seront déclarés n'importe où dans le texte. Si ces transferts externes (IX) sont hors blocs des familles internes, ils seront implicitement considérés comme des IX ; déclarés sous un bloc de famille interne, ils devront être explicitement déclarés externes. Notons que la cellule éventuellement sous Universe (U) n'est connectée qu'aux IX tranferts, et n'a ainis d'autre particularité que d'être sous U.

1.1 Pistes amonts de recherche

De la remarque de **Pat** considérant que Mortran n'a de comparable que le langage de programmation de TeX / LaTeX, il serait bon d'en étudier les principes, qui semblent issus ou cohérent avec le Lambda-calcul. Ceci afin, soit de situer, soit de remplacer Mortran dans un projet avancé. Nous resterons bien sùr ici dans notre langage familier, et le résultat pourra constituer une maquette pour interroger d'autres langages (dont peut-être Lisp, **JYG** dixit).

2 Résumé des calculs nécessaires

2.1 Calculs de trajectoire

Le principe de la 200 est celui du TEF-ZOOM en ne conservant qu'une cellule et qu'un transfert interne par famille, avec, ainsi, un seul niveau d'emboîtement des familles sous la famille englobante Universe.

Un système est ainsi défini pour chaque famille fm_i[1] :

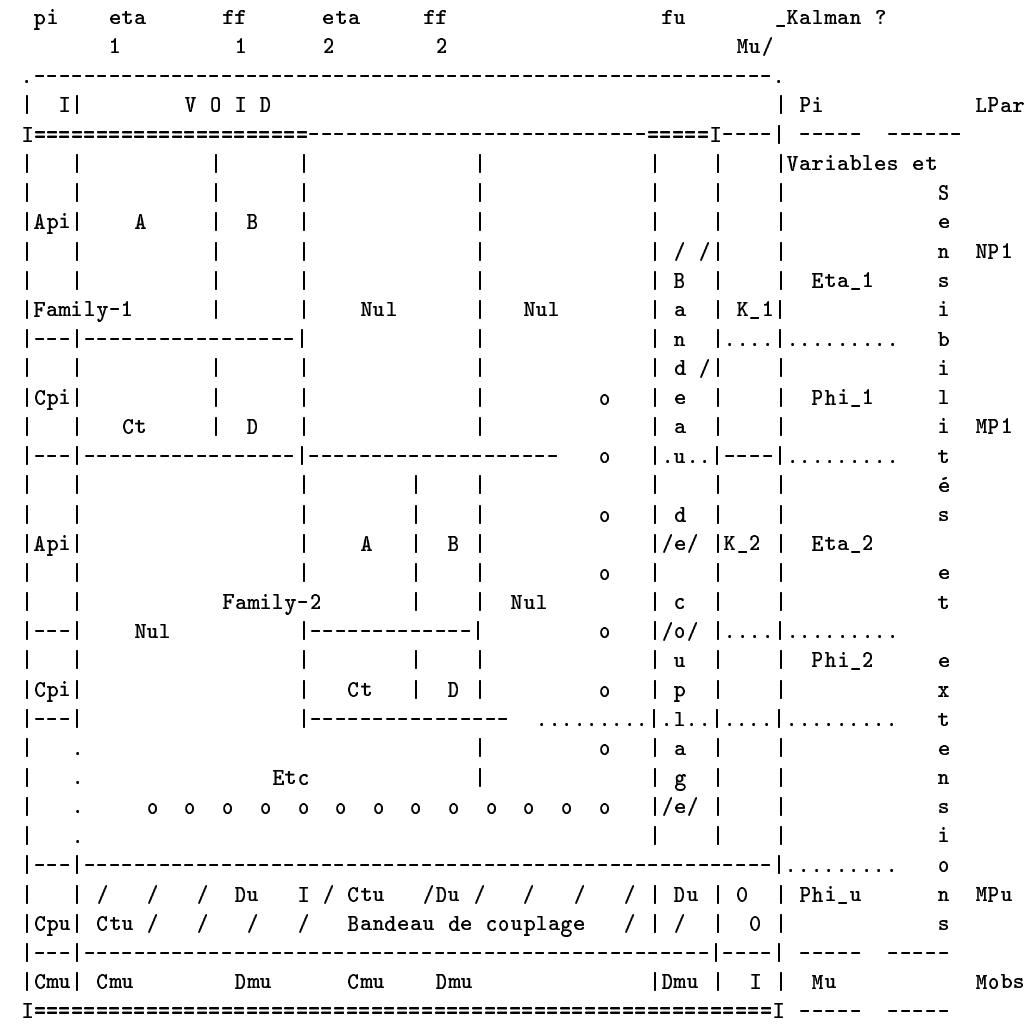
$$\begin{cases} \partial_t \eta_i(t) &= g_i(\eta_i(t), \varphi_i(t), \varphi_u, t) \\ \varphi_i(t) &= f_i(\eta_i(t), \varphi_i(t), \varphi_u, t) \end{cases}$$

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

$$\begin{bmatrix} \partial_t \delta\eta_i \\ 0 \end{bmatrix} = \begin{bmatrix} H_i & Bb_i & Bb_i^u \\ C_i^\dagger & D_i & D_i^u \end{bmatrix} \begin{bmatrix} \delta\eta_i \\ \delta\varphi_i \\ \delta\varphi_u \end{bmatrix} + \begin{bmatrix} \Gamma_i \\ \Omega_i \end{bmatrix} \tag{1}$$

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

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



[1] aucun rapport avec les éléphants.

avec les nouvelles matrices dépendant de δt et des Jacobiennes : $A = I - \frac{\delta t}{2} H$ et $B = -\frac{\delta t}{2} B_b$. On a d'emblée intégré les paramètres ϖ et les observables μ de la 102.

La résolution suit toujours les principes du TEF en appliquant une première étape d'élimination des variables $\delta\eta_i$, qui donne le système dit "de couplage" (entre les transferts internes et l'unique externe) :

$$[\ I - D_i + C_i^\dagger A_i^{-1} B_i\ ]\ \delta\varphi_i + [C_i^{\dagger u} A_i^{-1} B_i^u]\ \delta\varphi_u = \Omega_i + C_i^\dagger A_i^{-1} \Gamma_i \qquad (2)$$

La matrice $\mathcal{C} = [\ I - D + C^\dagger A^{-1} B\ ]$ qui est dite " de couplage interne", et le vecteur $\delta\varphi_{i,ins} = \Omega_i + C_i^\dagger A_i^{-1}\Gamma_i$ représente l'évolution insensible du transfert interne à chaque famille fm_i.

La deuxième étape consiste à éliminer les tranferts internes à partir des équations (1) et aboutir à la matrice de couplage entre les composantes du transfert externe.

Après résolution du système linéaire résultant, diît "sous U", on a $\delta\varphi_u$, et il reste à calculer les $\delta\varphi_i$ avec les systèmes (2) dans un algorithme dît de **descente** de familles.

Ceci étant fait, il reste à avancer les état : $\delta\eta_i = \Gamma_i + A_i^{-1} B_i \delta\varphi_i + A_i^{-1} B_i^u \delta\varphi_u$, ce qui finit de déterminer l'avance du système en dt.

L'état suivant $\eta_i(t+\delta t) = \eta_i(t) + \delta\eta_i$ étant connu, on détermine le nouveau vecteur des transferts par la résolution du système implicite $\varphi_i(t) = f_i(\eta_i(t), \varphi_i(t), \varphi_u(t), t)$ dans l'algorithme dît de "**la D_loop**".

2.2 Principes retenus, aspects matriciels et rangement

Le principe de génération du code consiste à utiliser l'étape **$DoDimEtaPhi** pour établir des listes de matrices de symboles suffixés par un numérotage de familles :

```
$dim$: real a(np,np+LPar),b(np,mp+MPu),Ct(MP,NP+LPar),D(MP,MP+MPu);
real Au(NPu,MPu),Bu(NPu,MPu),Ctu(MPu+Mobs,NPtot+LPar),Du(MPu+Mobs,MPtot);
$dim$: real AmB(np,mp+MPu);

! Pointeurs :
! liste de ko1lo == lofm(1)
!          ko1dim == kodimfm(1) avec nfm_int+1 == U
! NPtot et MPtot = NP1+NP2 + ... +Npu
$dim$: dimension noint(np),moint(mp);
!%% il faut en plus :
!+==================
equivalence (no1,nodim(1)),(mo1,modim(1));
integer nodim(4),modim(4),knoptr(4),kmoptr(4);
common/etaphidim/non,mom,knoptr,kmoptr;
```

qui devra générer, pour l'exemple donné :

```
      PARAMETER (NOF=2)
      PARAMETER (NP=4,MP=6)
      PARAMETER (LPar=3)
      PARAMETER(N=NP,M=MP)
      DIMENSION ETA(N),FF(M+LP),SENS(M+LP)
C LISTE GENEREE = (1,2)
      REAL A1(NP1,NP1+LPAR),B1(NP1,MP1+MPU),CT1(MP1,NP1+LPAR),D1(MP1,MP1
     *+MPU)
      REAL A2(NP2,NP2+LPAR),B2(NP2,MP2+MPU),CT2(MP2,NP2+LPAR),D2(MP2,MP2
     *+MPU)
      REAL AU(NPU,MPU),BU(NPU,MPU),CTU(MPU+MOBS,NPTOT+LPAR),DU(MPU+MOBS,
```

```
     *MPTOT)
C LISTE GENEREE = (1,2)
      REAL AMB1(NP1,MP1+MPU)
      REAL AMB2(NP2,MP2+MPU)
C LISTE GENEREE = (1,2)
      DIMENSION NOINT1(NP1),MOINT1(MP1)
      DIMENSION NOINT2(NP2),MOINT2(MP2)
      EQUIVALENCE(NO1,NODIM(1)),(MO1,MODIM(1))
      INTEGER NODIM(4),MODIM(4),KNOPTR(4),KMOPTR(4)
      COMMON/ETAPHIDIM/NON,MOM,KNOPTR,KMOPTR
```

On reconnait NOF familles, quatre états et six transferts en tout, Lpar paramètres etc Les dimensions des blocs de vecteurs et matrices sont du type NOxxx(), MOxxx(), rassemblant les NPi,..., MPi. Tout objet sous U est suffixé par "u". De même pour les pointeurs de type KNOxxx, KMOxxx.

On comprend avec ce brouillon le principe visant :

- à remplacer les déclarations de la 102 par une $dim générant une liste de déclarations ;
- à calculer des pointeurs rangés en tableaux Fortran ;
- à partir de celà, un appel à sous-programme permettra d'itérer les calculs par famille en pointant les matrices et vecteurs rangés consécutivement en tableaux ;

2.2.1 Jacobiennes

Le principe de calcul des Jacobiennes est pour l'instant[1] le suivant :

```
!-----------------------------------
! matrices Ct(i,j) = dFi(i) / dEta(j)
!-----------------------------------
      do kof=1,nof
      < do j=1,nodim(kof)
        < mult[i]=1,mp : Ct([.i],j) = f_d(Phi_tef[i])(/eta(j));
        >;
      >;
! partie parametres
!                      dFi(i) / dPi_j
! -----------------------------------
       if lp.gt.0
        < mult [i]=1,mp: mult[j]=1,lp:
          Ct([.i],np+[.j]) = F_D(Phi_tef[i])(/eta(np+[.j]));
        >;
```

où on voit que la structure de la 102 est essentiellement conservée en indiçant les dimensions. En particulier, le code de dérivation est inchangé, mais le $ZINIT associe aux noms de variables (uniques) un suffixe qui est l'ordinal de sa famille et un critère de parcours, si bien que le code généré est :

```
10012 DO 10015 KOF=1,NOF
      DO 10017 J=1,NODIM(KOF)
      IF(KOF.EQ.1) CT1(1,J)=(F_DELTA(1,J))     * * *
      IF(KOF.EQ.2) CT2(2,J)=((F_DELTA(3,J))*(ETA(1))+(F_DELTA(1,J))*(ETA
     *(3)))
```

[1] on a choisit dans ce texte de conserver les itérations vers la solution finale – toujours sans aucun rapport avec les éléphants – et l'option décrite ici a été abandonnée comme on le verra.

```
      CT( 1,J)=(( (F_DELTA(1,J))*(2.))+( (F_DELTA(3,J))*(0.)))
      CT( 2,J)=((F_DELTA(1,J))+(F_DELTA(3,J)))
10017 CONTINUE
10018 CONTINUE
10015 CONTINUE
10016 IF(.NOT.( LP.GT.0))GOTO 10019
      IF(KOF.EQ.1) CT1(1,NP+1)=(0.)
      * * *
      IF(KOF.EQ.2) CT2(2,NP+3)=( (0.))         * * *
      CT( 1,NP+1)=(( (0.))+( (0.)))
      CT( 2,NP+3)=( (0.))
```

Ainsi, chaque matrice (ici la $C^{\dagger}$) n'est remplie qu'au parcours de sa famille, sinon on déborde le tableau matriciel.

Les objets sous U conservent leur nom dans la 102. On peut penser à donner comme ordinal de U NOF+1 et y associer un critère semblable aux internes. On a donné d'emblée la partie paramètres.

On peut craindre que l'introduction systématique d'un critère logique ralentisse le parcours de boucle mais je n'est pas encore trouvé d'alternative. Ce pb est accru par l'introduction de familles avec grille, que l'on peut imaginer ainsi :

```
l'écriture dans la 102 :
++++++++++++++++++++++++
!-----------------------------------
! matrices Ct(i,j) = dFi(i) / dEta(j)
!-----------------------------------
        if m_dwn .gt.0
        < do j=1,np
          < ;mult [i]=1,m_dwn :    Ct([.i],j) = F_D(Phi_tef[i])(/eta(j));
          >;
        >;
        do j=1,np
        < do inode=1,m_node
          < ;mult[i]=1,m_mult :
             Ct(k2j_[.i](inode),j) = F_D(nde_ff_fun_[.i](inode))(/eta(j));
          >;
        >;
        if m_up .gt.0
        < do j=1,np
          < ;mult [i]=1,m_up :
                    Ct(joffsetup+[.i],j) = F_D(Phi_tef(up[.i]))(/eta(j));
          >;
        >;
! partie Obs
        if mobs.gt.0
        < do j=1,np
          < ;mult [i]=1,mobs :
                    Ct(mp+[.i],j) = F_D(Phi_tef(mp+[.i]))(/eta(j));
          >;
        >;

l'écriture dans la 200 :
++++++++++++++++++++++++
```

```
!-----------------------------------
! matrices Ct(i,j) = dFi(i) / dEta(j)
!-----------------------------------
      do kof=1,nof
      < do j=1,nodim(kof)
        < do inode=1,m_node(kof)
          <
            mult[i]=1,mp : Ct([.i],j) = f_d(Phi_tef[i])(/eta(j));
          >
        >;
      >;
! partie parametres
!                    dFi(i) / dPi_j
! --------------------------------
       if lp.gt.0
        < mult [i]=1,mp: mult[j]=1,lp:
          Ct([.i],np+[.j]) = F_D(Phi_tef[i])(/eta(np+[.j]));
        >;
```

c'est-à-dire que l'on a une macro dans set_node_phi qui associe la Ct à sa forme (inode), soit en bouclant aussi sur les dwn et up, soit en y ajoutant le critère if (inode.eq.1). Du coup on aurait un m_node qui engloberait les dwn et up, disons un mp_i qui ne colle plus à la dimension des vecteurs. Le symbole Phi_tef[i] est alors standard. Les up restent marqués par up[.i] et les obs par (mp+[.m]. Le Fortran généré ressemblearit à ça :

```
10117 DO 10120 kof=1,nof
      DO 10121 j=1,nodim(kof)
      DO 10123 inode=1,m_node(kof)
C les down meshes
      if (kof.eq.1.and.inode.eq.1) Ct(1,j)=( ((((F_delta(k2i_1(1),j))*(e
     *ta(k2i_3(1)))+(F_delta(k2i_3(
     *1),j))*(eta(k2i_1(1))))+((F_delta(1,j))*(eta(3))+(F_delta(3,j))*(e
     *ta(1))))/(ff(9))-(0.)*2)*(0.5))
      if (kof.eq.1.and.inode.eq.1) Ct(2,j)=( (-(0.)*2)*(ff(1)))
 . . .
C les node meshes
      if (kof.eq.1) Ct(k2j_1(inode),j)=( ((((F_delta(k2i_1(inode+1),j))*(eta(k2i_3(ino
     *de+1)))+(F_delta(k2i_3(inode+1),j))*(eta(k2i_1(inode+1))))+((F_del
     *ta(k2i_1(inode),j))*(eta(k2i_3(inode)))+(F_delta(k2i_3(inode),j))*
     *(eta(k2i_1(inode)))))/(ff(k2j_9(inode)))-(0.)*2)*(0.5))
 . . .
      Ct(k2j_2(inode),j)=((( (0.))-( (0.)))/(dz(inode))-(0.)*2)
      Ct(k2j_5(inode),j)=( (((F_delta(k2i_2(inode),j))-(0.))/(theta_e(in
     *ode))-(0.)*2)*(gg))
 . . .
C les up meshes comme les dwn
10123 CONTINUE
10124 CONTINUE
10121 CONTINUE
10120 CONTINUE
```

Ainsi, la **mult** génère toutes les lignes associées aux fonctions différentes, et c'est via les macros transformant les Ct() que l'on peut faire la distinction entre mailles et simples. Il faut donc alouter n_node(1)=1 et m_node aux simples.

solution ? On va (probalement) opter pour un autre principe qui est basé sur ce que la 200 oblige à faire, à savoir une association quasi systématique entre les lignes diverses de calcul du TEF générées par la surmacro **mult :** et les variables Fortran. Rappellons que, *in fine*, à chaque création d'objet cellule-transfert est associé un numéro d'ordre (dans le stack 5 pour les états, 6 pour les transferts). Alors, toute composante de vecteur ou ligne de matrice est repérée par cet ordinal. On peut donc à l'envi modifier les noms de variables Fortran, particulariser chaque famille etc.

On a à identifier les quatre Jacobiennes et les vecteurs **eta(), dot_eta()**, pour obtenir par exemple :

```
      FF(1)=(ETA(1))
      FF(2)=(ETA(2))
      FF(3)=(2.*ETA(1)+3.*ETA(2)+4.*ETA(K2I_1(1)))
      DO INODE=1,MNODE(2)
      FF(K2J_1(INODE))=((ETA(K2I_1(INODE))*ETA(2)))
      FF(K2J_2(INODE))=((ETA(K2I_2(INODE))*FF(K2J_1(INODE))))
      ENDDO
      FF(6)=(ETA(1)+ETA(K2I_1(3))+11.*ETA(K2I_2(3)))
      DOT_ETA(1)=(PI_PRANDTL*FF(2)-PI_PRANDTL*ETA(1))
      DOT_ETA(2)=(-10.+PI_RAYLEIGH_RATIO*FF(1)-ETA(2))
      DOT_ETA(3)=(0.)
      DOT_ETA(4)=(0.)
      DO INODE=1,NNODE(2)
      DOT_ETA(K2I_1(INODE))=((FF(K2J_1(INODE))-PI_WAVE_NB_RATIO*ETA(K2I_
     *1(INODE))))
      DOT_ETA(K2J_2(INODE))=((FF(K2J_2(INODE))-PI_WAVE_NB_RATIO*ETA(K2I_
     *2(INODE))))
      ENDO
      DOT_ETA(7+IOFUP2)=(0.)
      DOT_ETA(8+IOFUP2)=(0.)
      DOT_ETA(9)=(45.)
C ===== FAMILY-1 =====
      DO J=1,MP1
      D1(1,J)=(0.)
      D1(2,J)=(0.)
      ENDDO
C ===== FAMILY-U =====
      DO J=1,MP
      D( 2,J)=(( (0.))+(( (0.))+( (0.))))
      ENDDO
C ===== FAMILY-2 =====
      DO J=1,MP2
      DO INODE=1,MNODE(2)
      D2(K2J_1(INODE),J)=( (0.))
      D2(K2J_2(INODE),J)=( (F_DELTA(K2J_1(INODE),J))*(ETA(K2I_2(INODE)))
     *)
      ENDDO
      ENDDO
C ===== FAMILY-U =====
      DO J=1,MP
      D( 3,J)=( ( ( (0.))))
      ENDDO
```

Dans cet exemple, la deuxième famille est maillée avec une partie **Dwn** et **Up**. Comme on le voit, il suffit de repérer un début et fi9jn de famille pour générer les débuts et fins de boucles — sur la variable de dérivation, sur un nonbre de mailles ...

En conséquence, les instructions **mult** sont simplifiées, ici :

```
!---------------------------
! Calcul de ff = f(eta,Phiz)
!---------------------------
  ;mult [j]=1,mp : ff[j] = Phi_tef[j];
!-------------------------------
! Calcul de dot_eta = g(eta,Phi)
!-------------------------------
   ;mult[i]=1,np : dot_eta[i] = deta_tef[i];
!-------------------------------
! Matrice d(i,j) = dFi(i)/dFi(j)
!-------------------------------
   ;mult [i]=1,mp : d([.i],j) = f_d(Phi_tef[i])(/ff(j));
```

La conversion précédente est incomplète car elle ne tiend pas compte des décalages introduits par une famille maillée sur les suivantes. Pour concrétiser ce problème, et dans le même temps produire un code Fortran explicite, on peut marquer y compris les variables vectorielles par leur famille : **ff1(), dot_eta1())**, en laissant aux macros de la séquence **$DoDimEtaPhi** le soin de calculer les **ioffset**. On aurait alors :

```
      FF1(1)=(ETA1(1))
      FF1(2)=(ETA1(2))
      FF(3)=(2.*ETA1(1)+3.*ETA1(2)+4.*ETA2(K2I_1(1)))
      DO INODE=1,MNODE(2)
      FF2(K2J_1(INODE))=((ETA2(K2I_1(INODE))*ETA1(2)))
      FF2(K2J_2(INODE))=((ETA2(K2I_2(INODE))*FF2(K2J_1(INODE))))
      ENDDO
      FF(6)=(ETA1(1)+ETA2(K2I_1(3))+11.*ETA2(K2I_2(3)))
      WRITE(*,*)'   FF :'
      WRITE(*,*)'   ',FF
      DOT_ETA1(1)=(PI_PRANDTL*FF1(2)-PI_PRANDTL*ETA1(1))
      DOT_ETA1(2)=(-10.+PI_RAYLEIGH_RATIO*FF1(1)-ETA1(2))
      DOT_ETA2(1)=(0.)
      DOT_ETA2(2)=(-10.+PI_RAYLEIGH_RATIO*FF1(1)-ETA1(2))
      DO INODE=1,NNODE(2)
      DOT_ETA2(K2I_1(INODE))=((FF2(K2J_1(INODE))-PI_WAVE_NB_RATIO*ETA2(K
     *2I_1(INODE))))
      DOT_ETA2(K2J_2(INODE))=((FF2(K2J_2(INODE))-PI_WAVE_NB_RATIO*ETA2(K
     *2I_2(INODE))))
      ENDO
      DOT_ETA2(IOFUP2+1)=(0.)
      DOT_ETA2(IOFUP2+2)=(-10.+PI_RAYLEIGH_RATIO*FF1(1)-ETA1(2))
      DOT_ETA3(1)=(45.)
```

On remarque qu'on a gardé le tableau standard **FF()** pour marquer les transferts sous **U** qui jouent un rôle particulier — ce qui n'est pas le cas de la cellule sous **U**. L'intérêt de cette forme est que l'on détecte dans le code Fortran une anomalie de connection — ici par exemple, FF(3) est sous **U** et peu donc être connecté a tout objet, mais pas FF2 qui utilise la cellule de la première famille.

problème Oui, car ainsi, si on souhaite garder la simplicité de la **mult**, on dérive, mettons, **FF1()** par FF() ! On se rabbat donc sur la forme FF(), mais comme il faut indicer avec un zéro-pointeur à cause de l'intervention des familles maillées, on considère que la Fortran reste lisible avec ce marquage famillial :

```
      FF(JO1+1)=(ETA(IO1+1))
      FF(JO1+2)=(ETA(IO1+2))
      FF(JOU+2)=(2.*ETA(IO1+1)+3.*ETA(IO1+2)+4.*ETA(IO2+K2I_1(1)))
      DO INODE=1,MNODE(2)
      FF(2+K2J_1(INODE))=((ETA(IO2+K2I_1(INODE))*ETA(IO1+2)+41.*FF(JOU+3
     *)))
      FF(JO2+K2J_2(INODE))=((ETA(IO2+K2I_2(INODE))*FF(JO2+K2J_1(INODE)))
     *)
      ENDDO
      FF(JOU+3)=(ETA(IO1+1)+ETA(IO2+K2I_1(3))+11.*ETA(IO2+K2I_2(3)))
      WRITE(*,*)'   FF :'
      WRITE(*,*)'   ',FF
      DOT_ETA(IO1+1)=(PI_PRANDTL*FF(JO1+2)-PI_PRANDTL*ETA(IO1+1))
      DOT_ETA(IO1+2)=(-10.+PI_RAYLEIGH_RATIO*FF(JO1+1)-ETA(IO1+2))
      DOT_ETA(IO2+1)=(FF(JOU+2))
      DOT_ETA(IO2+2)=(-10.+PI_RAYLEIGH_RATIO*FF(JO1+1)-ETA(IO1+2))
      DO INODE=1,NNODE(2)
      DOT_ETA(2+K2I_1(INODE))=((FF(JO2+K2J_1(INODE))-PI_WAVE_NB_RATIO*ET
     *A(IO2+K2I_1(INODE))))
      DOT_ETA(IO2+K2I_2(INODE))=((FF(JO2+K2J_2(INODE))-PI_WAVE_NB_RATIO*
     *ETA(IO2+K2I_2(INODE))))
      ENDO
      DOT_ETA(IO2+IOFUP2+1)=(0.)
      DOT_ETA(IO2+IOFUP2+2)=(-10.+PI_RAYLEIGH_RATIO*FF(JO1+1)-ETA(IO1+2)
     *)
      DOT_ETA(IO3+1)=(45.)
```

où on a noté ces zéro-pointeurs par jo pour les transferts et io pour les états.

Cette solution s'étend aux Jacobiennes :

```
!------------------------------
! Matrice d(i,j) = dFi(i)/dFi(j)
!------------------------------
          ;mult [i]=1,mp : d([.i],j) = f_d(Phi_tef[i])(/ff(j));
 qui donne :

C ===== FAMILY-1 =====
      DO J=1,MP1
      D1(1,J)=(0.)
      D1(2,J)=(0.)
      ENDDO
C ===== FAMILY-U =====
      DO J=1,MP
      D( 2,J)=(( (0.))+(( (0.))+( (0.))))
      ENDDO
C ===== FAMILY-2 =====
      DO J=1,MP2
      DO INODE=1,MNODE(2)
```

```
      D2(K2J_1(INODE),J)=(( (0.))+( (F_DELTA(JOU+3,JO2+J))*(41.)))
      D2(K2J_2(INODE),J)=( (F_DELTA(JO2+K2J_1(INODE),JO2+J))*(ETA(IO2+K2
     *I_2(INODE))))
      ENDDO
      ENDDO
C ===== FAMILY-U =====
      DO J=1,MP
      D( 3,J)=( ( ( (0.))))
      ENDDO

!-------------------------------------------------------
! matrice H(i,j) = dEta(i)/dEta(j)
!-------------------------------------------------------
          mult[i]=1,np : H([.i],j) = f_d(deta_tef[i])(/eta(j));
qui donne :

C ===== FAMILY-1 =====
      DO J=1,NP1
      H1(1,J)=(( (0.))-( (F_DELTA(IO1+1,JO1+J))*(PI_PRANDTL)))
      H2(IOFUP2+2,J)=(((-(0.))+( (0.)))+(-(F_DELTA(IO1+2,JO1+J))))
C ===== FAMILY-2 =====
      DO J=1,NP2
      H2(1,J)=(0.)
      ENDDO
C ===== FAMILY-2 =NDE=
      DO J=1,NP2
      DO INODE=1,NNODE(2)
      H2(K2I_1(INODE),J)=(-( (F_DELTA(IO2+K2I_1(INODE),JO1+J))*(PI_WAVE_NB_R
     *ATIO)))
      H2(K2I_2(INODE),J)=(-( (F_DELTA(IO2+K2I_2(INODE),JO1+J))*(PI_WAVE_NB_R
     *ATIO)))
      ENDO
      ENDDO
C ===== FAMILY-2 =UP=
      DO J=1,NP2
      H2(IOFUP2+1,J)=(0.)
      ENDDO
C ===== FAMILY-3 =====
      DO J=1,NP3
      H3(1,J)=(0.)
      ENDDO
```

où on met l'accent sur les indices :

- les matrices sont séparées, les indices sont intra-famille ;
- les fonhctions d'indices de maille sont du type **k[fm]i_[k]** avec un premier marqueur celui de la famille et un deuxième de l'ordinal dans la famille (même chose pour les transferts avec **k[fm]j_[k]**) ;
- les offset **IOFUP[fm]** sont internes et relatifs au nombre de mailles et à leur multiplicité ;
- la dérivation se fait par rapport **FF(jo[fm]+J)**, c'est-à-dire la position de la composante de transfert dans le vecteur FF() global ;
- on remarque le cas particulier des transferts externes (sous U), qui sont dérivés y compris par rapport à tous les autres ;

– on rapelle que la cellule “externe” est une cellule sans prérogatives, elle est en réalité une “cellule interne à U” ;

De plus, par rapport à la 102, on a ici des transferts externes que chaque objet (interne ou externe) doit aussi prendre en compte. Il faut donc une deuxième série de dérivation pour calculer les matrices Bb() et D(). On choisit de marquer cette deuxième dérivation par F_D(#)(/FF(JOU+J) ; on suppose en effet que les composantes du transfert externe sont rangées en fin du tableau FF() et sont en nombre **MPU**. On a alors :

```
!------------------------------
! Matrice d(i,j) = dFi(i)/dFi(j)
!------------------------------
! Il faut egalement deriver / ix_Phi
! ----------------------------------
          ;mult [i]=1,mp : d([.i],j) = f_d(Phi_tef[i])(/ff(jou+j));
ce qui ajoute~:

C ===== FAMILY-1 =====
      DO J=1,MPU
      D1(1,MP1+J)=(0.)
      D1(2,MP1+J)=(0.)
      ENDDO
      ENDDO
C ===== FAMILY-2 =====
      DO J=1,MPU
      DO INODE=1,MNODE(2)
      D2(K2J_1(INODE),MP2+J)=(( (0.))+( (F_DELTA(JOU+3,JOU+J))*(41.)))
      D2(K2J_2(INODE),MP2+J)=( (F_DELTA(JO2+K2J_1(INODE),JOU+J))*(ETA(IO
     *2+K2I_2(INODE))))
      ENDDO
      ENDDO
      ENDDO
```

On remarque que cela n’a pas affecté les transferts externes qui sont déjà tous calculés lors de la première série.

sensibilité aux paramètres. On garde essentiellement la même forme de calculs que pour la 102, mais on associe le symbole oPi[i] aux paramètres.

2.3 État des lieues au 12 Novembre 2007

S'il reste encore du chemin à courir, on peut s'arrêter un moment sur la "solution" actuelle.

- cellules internes ou sous U semblables et restrictives dans le sens ou on a soit un set_eta, soit trois blocs au plus de nodes ;
- idem pour les transferts internes ;
- par contre, le transfert sous U jouit d'extraordinaires propriétés, pour atteindre la souplesse de connection requise : on peut enchaîner des set_phi et dwn_phi, node_phi et up_phi à peu prés comme on veut, et ceci en tout lieu. Cette souplesse permet a la fin de chaque création de famille interne de régler ses connections globales ;
- le rangement des matrices jacobiennes est fait en prolongeant :
  - H_i et Ct_i des dérivées par rapport aux paramètres ;
  - Bb_i et D_i des dérivées par rapport aux transferts externes ;
  - les matrices du transfert externe gardent leur nom standard ;
  - de même pour les probes (ou mesures) ;
- il manque dans cette liste le statut des familles Kalman, **à toi de jouer, Pat** ;

Pour illustrer l'état de l'art deucents, voir cet exemple, sans aucune signification physique assumée, en commençant par le ZINIT :

```
Under Universe
< UNDER Part-1
  <
! Cell definition
    set_eta
    < var: eta_courant_L,
      fun: deta_conv = pi_prandtl*ff_T_czcx
                                    - pi_prandtl*eta_courant_L;
      var: eta_T_czcx,
      fun: deta_energy_zx = - 10. + pi_Rayleigh_ratio*ff_courant_L
                                                    - eta_T_czcx;
   >;
! Transfer definition
    set_Phi
    <
      eqn: ff_courant_L  = eta_courant_L;       "x"
      eqn: ff_T_czcx     = eta_T_czcx;          "y"
    >;
  >;"end Fm"
;
 UNDER Part-2
 <
! Cell definition
  Set_DWN_Eta
  <   eqn: eta_T_sz(0) = ff_bid_X;
      eqn: eta_T_bb(0) = 0.0;
  >;
  IN_[3]eta
  <
    var: eta_T_sz(.),
    fun: deta_energy_z(inode) = ff_Psi_Tsz(inode) - pi_wave_nb_ratio*eta_T_sz(inode);
    eqn: eta_T_bb(inode)= ff_Psi_Tbb(inode) - pi_prandtl*eta_T_bb(inode);
  >;
```

```
;
  Set_UP_Eta
  <   eqn: eta_T_sz(Ngrid) = 0.1;
      eqn: eta_T_bb(Ngrid) = 0.2;
  >;
! *** irruption d'un IX-Tr
    IX_Phi
    < eqn: ff_bid_X =  2.*eta_courant_L + 3.*eta_T_czcx
                       +4.*eta_T_sz(inode:1);
    >;
! Transfer definition
   DWN_PHI
   <  eqn: ff_Psi_Tczcx(0) = eta_T_sz(0);
   >;
   IN_[3]Phi
   <
     var: ff_Psi_Tsz(.), fun: ggg(.)= eta_T_sz(inode)*eta_T_czcx +41.*ff_bid_Y(inode);
     var: ff_Psi_Tbb(.), fun: ddd(.) = eta_T_bb(inode)*ff_Psi_Tsz(inode);
   >;
   UP_PHI
   <  eqn: ff_Psi_Tczcx(up) = eta_T_sz(inode:3);
   >;
 >;"end Fm"
! *** irruption d'un IX-Tr maillé
  DWN_IX_PHI
  < eqn: ff_bid_Z(0) = dwn*ff_Psi_Tczcx(0);
  >;
  IX_[3]Phi
  < eqn: ff_bid_Y(.) =  eta_courant_L + eta_T_sz(inode) + 11.*eta_T_bb(inode);
    eqn: ff_bid_Z(.) =  23.*ff_Psi_Tbb(inode);
  >;
  UP_IX_PHI
  < eqn: ff_bid_Y(4) = eta_T_sz(inode:3);
  >;
  set_Eta                "la cellule sous U"
  < eqn: eta_bid=45.;
  >;
>;"End Universe"
```

avec des morceaux choisis de ce que ça devrait donner, d'abord pour les vecteurs :

```
!---------------------------
! Calcul de ff = f(eta,Phiz)
!---------------------------
;mult [j]=1,mp : ff[j] = Phi_tef[j];
      FF(JO1+1)=(ETA(IO1+1))
      FF(JO1+2)=(ETA(IO1+2))
      FF(JOU+ 1)=(2.*ETA(IO1+1)+3.*ETA(IO1+2)+4.*ETA(IO2+K2I_1(1)))
      FF(JO2+1)=(ETA(IO2+1))
      DO INODE=1,MNODE(2)
      FF(JO2+K2J_1(INODE))=((ETA(IO2+K2I_1(INODE))*ETA(IO1+2)+41.*FF(JOU
     *+KUJ_3(INODE))))
      FF(JO2+K2J_2(INODE))=((ETA(IO2+K2I_2(INODE))*FF(JO2+K2J_1(INODE)))
```

```
     *)
      ENDDO
      FF(JO2+JOFUP2+1)=(ETA(IO2+K2I_1(3)))
      FF(JOU+ 2)=(DWN*FF(JO2+1))
      DO INODE=1,MNODE(NOF+1)
      FF(JOU+KUJ_3(INODE))=((ETA(IO1+1)+ETA(IO2+K2I_1(INODE))+11.*ETA(IO
     *2+K2I_2(INODE))))
      FF(JOU+KUJ_4(INODE))=((23.*FF(JO2+K2J_2(INODE))))
      ENDDO
      FF(JOU+JOFUPU+5)=(ETA(IO2+K2I_1(3)))
      WRITE(*,*)'   FF :'
      WRITE(*,*)'   ',FF
!------------------------------
! Calcul de dot_eta = g(eta,Phi)
!------------------------------
 ;mult[i]=1,np : dot_eta[i] = deta_tef[i];
      DOT_ETA(IO1+1)=(PI_PRANDTL*FF(JO1+2)-PI_PRANDTL*ETA(IO1+1))
      DOT_ETA(IO1+2)=(-10.+PI_RAYLEIGH_RATIO*FF(JO1+1)-ETA(IO1+2))
      DOT_ETA(IO2+1)=(FF(JOU+ 1))
      DOT_ETA(IO2+2)=(0.0)
      DO INODE=1,NNODE(2)
      DOT_ETA(IO2+K2I_1(INODE))=((FF(JO2+K2J_1(INODE))-PI_WAVE_NB_RATIO*
     *ETA(IO2+K2I_1(INODE))))
      DOT_ETA(IO2+K2I_2(INODE))=((FF(JO2+K2J_2(INODE))-PI_PRANDTL*ETA(IO
     *2+K2I_2(INODE))))
      ENDDO
      DOT_ETA(IO2+IOFUP2+1)=(0.1)
      DOT_ETA(IO2+IOFUP2+2)=(0.2)
      DOT_ETA(IOU+1)=(45.)
```

où on voit le rôle des io ou jo pointeurs dans le tableau des vecteurs eta(.) et ff(.), alors que les matrices sont associées à chaque famille - et indicées en rapport :

```
!------------------------------
! Matrice d(i,j) = dFi(i)/dFi(j)
!------------------------------
          ;mult [i]=1,mp : d([.i],j) = f_d(Phi_tef[i])(/ff(j));
C ===== FAMILY-1 == TR  =====
      DO J=1,MP1
      D1(1,J)=(0.)
      D1(2,J)=(0.)
      ENDDO
C ===== FAMILY-U == MU  =====
      DO J=1,MP
      D(MPU+1,J)=( ( (0.)))
      ENDDO
C ===== FAMILY-U == TR  =====
      DO J=1,MP
      D( 1,J)=(( (0.))+(( (0.))+( (0.))))
      ENDDO
C ===== FAMILY-2 == DWN-TR =====
      DO J=1,MP2
      D2(1,J)=(0.)
```

```
      ENDDO
C ===== FAMILY-2 == NDE-TR ===
      DO J=1,MP2
      DO INODE=1,MNODE(2)
      D2(K2J_1(INODE),J)=(( (0.))+( (F_DELTA(JOU+KUJ_3(INODE),JO2+J))*(4
     *1.)))
      D2(K2J_2(INODE),J)=( (F_DELTA(JO2+K2J_1(INODE),JO2+J))*(ETA(IO2+K2
     *I_2(INODE))))
      ENDDO
      ENDDO
C ===== FAMILY-2 == UP-TR =====
      DO J=1,MP2
      D2(JOFUP2+1,J)=(0.)
      ENDDO
C ===== FAMILY-U == MU  =====
      DO J=1,MP
      D(MPU+2,J)=(F_DELTA(JO2+JOFUP2+1,J))
      ENDDO
C ===== FAMILY-U == DWN-TR =====
      DO J=1,MP
      D( 2,J)=( (F_DELTA(JO2+1,J))*(DWN))
      ENDDO
C ===== FAMILY-U == NDE-TR ===
      DO J=1,MP
      DO INODE=1,MNODE(NOF+1)
      D(KUJ_3(INODE),J)=( ( ( (0.))))
      D(KUJ_4(INODE),J)=( (F_DELTA(JO2+K2J_2(INODE),J))*(23.))
      ENDDO
      ENDDO
C ===== FAMILY-U == UP-TR =====
      DO J=1,MP
      D(JOFUPU+5,J)=(0.)
      ENDDO
C ===== FAMILY-U == MU  =====
      DO J=1,MP
      D(MPU+3,J)=( ( (F_DELTA(JOU+JOFUPU+5,J))*(ETA(IO2+K2I_1(3))))*(ETA
     *(IO1+1)))
      ENDDO
! Il faut egalement deriver / ix_Phi
! ----------------------------------
          ;mult [i]=1,mp : d([.i],j) = f_d(Phi_tef[i])(/ff(jou+j));
C ===== FAMILY-1 == TR  =====
      DO J=1,MPU
      D1(1,MP1+J)=(0.)
      D1(2,MP1+J)=(0.)
      ENDDO
C ===== FAMILY-2 == DWN-TR =====
      DO J=1,MPU
      D2(1,MP2+J)=(0.)
      ENDDO
C ===== FAMILY-2 == NDE-TR ===
      DO J=1,MPU
```

```
      DO INODE=1,MNODE(2)
      D2(K2J_1(INODE),MP2+J)=(( (0.))+( (F_DELTA(JOU+KUJ_3(INODE),JOU+J)
     *)*(41.)))
      D2(K2J_2(INODE),MP2+J)=( (F_DELTA(JO2+K2J_1(INODE),JOU+J))*(ETA(IO
     *2+K2I_2(INODE))))
      ENDDO
      ENDDO
C ===== FAMILY-2 == UP-TR =====
      DO J=1,MPU
      D2(JOFUP2+1,MP2+J)=(0.)
      ENDDO
C ===== FAMILY-U == NDE-TR ===
      DO J=1,MPU
      DO INODE=1,MNODE(NOF+1)
      D(KUJ_3(INODE),MP+J)=( ( ( (0.))))
      D(KUJ_4(INODE),MPU+J)=( (F_DELTA(JO2+K2J_2(INODE),J))*(23.))
      ENDDO
      ENDDO

!-------------------------------------------------------
! matrice  H(i,j) = dEta(i)/dEta(j)
!-------------------------------------------------------
         mult[i]=1,np : H([.i],j) = f_d(deta_tef[i])(/eta(j));
C ===== FAMILY-1 == CL  =====
      DO J=1,NP1
      H1(1,J)=(( (0.))-( (F_DELTA(IO1+1,IO1+J))*(PI_PRANDTL)))
      H1(2,J)=(((-(0.))+( (0.)))+(-(F_DELTA(IO1+2,IO1+J))))
      ENDDO
C ===== FAMILY-2 == DWN-CL =====
      DO J=1,NP2
      H2(1,J)=(0.)
      H2(2,J)=(0.)
      ENDDO
C ===== FAMILY-2 == NDE-CL =====
      DO J=1,NP2
      DO INODE=1,NNODE(2)
      H2(K2I_1(INODE),J)=(-( (F_DELTA(IO2+K2I_1(INODE),IO2+J))*(PI_WAVE_
     *NB_RATIO)))
      H2(K2I_2(INODE),J)=(-( (F_DELTA(IO2+K2I_2(INODE),IO2+J))*(PI_PRAND
     *TL)))
      ENDDO
      ENDDO
C ===== FAMILY-2 == UP-CL  =====
      DO J=1,NP2
      H2(IOFUP2+1,J)=(0.)
      H2(IOFUP2+2,J)=(0.)
      ENDDO
C ===== FAMILY-U == CL  =====
      DO J=1,NPU
      HU(1,J)=(0.)
      ENDDO
! partie parametres libres
```

```
       if lp.gt.0
       <  mult[k]=1,lp: mult [i]=1,np:
             H([.i],np+[.k]) = F_D(deta_tef[i])(/oPi[k]);
       >;
          mult[i]=1,np : Bb([.i],j)= f_d(deta_tef[i])(/ff(jou+j));
      IF(.NOT.( LP.GT.0))GOTO 10013
      H1(1,NP+1)=(( (FF(JO1+2))+(0.))-( (ETA(IO1+1))+(0.)))
      H1(2,NP+1)=(((-(0.))+( (0.)))+(-(0.)))
      H2(1,NP+1)=(0.)
      H2(2,NP+1)=(0.)
      DO INODE=1,NNODE(2)
      H2(K2I_1(INODE),NP+1)=(-( (0.)))
      H2(K2I_2(INODE),NP+1)=(-( (ETA(IO2+K2I_2(INODE)))+(0.)))
      ENDDO
      H2(IOFUP2+1,NP+1)=(0.)
      H2(IOFUP2+2,NP+1)=(0.)
      HU(1,NP+1)=(0.)
      H1(1,NP+2)=(( (0.))-( (0.)))
      H1(2,NP+2)=(((-(0.))+( (FF(JO1+1))+(0.)))+(-(0.)))
      H2(1,NP+2)=(0.)
      H2(2,NP+2)=(0.)
      DO INODE=1,NNODE(2)
      H2(K2I_1(INODE),NP+2)=(-( (0.)))
      H2(K2I_2(INODE),NP+2)=(-( (0.)))
      ENDDO
      H2(IOFUP2+1,NP+2)=(0.)
      H2(IOFUP2+2,NP+2)=(0.)
      HU(1,NP+2)=(0.)
      H1(1,NP+3)=(( (0.))-( (0.)))
      H1(2,NP+3)=(((-(0.))+( (0.)))+(-(0.)))
      H2(1,NP+3)=(0.)
      H2(2,NP+3)=(0.)
      DO INODE=1,NNODE(2)
      H2(K2I_1(INODE),NP+3)=(-( (ETA(IO2+K2I_1(INODE)))+(0.)))
      H2(K2I_2(INODE),NP+3)=(-( (0.)))
      ENDDO
      H2(IOFUP2+1,NP+3)=(0.)
      H2(IOFUP2+2,NP+3)=(0.)
      HU(1,NP+3)=(0.)
10013 CONTINUE
```

où on peut voir le choix nouveau de prendre oPi[.] comme symbole des paramètres sensibles.

```
!-----------------------------------------
!  Bb = d(phi)G et B = -dt/2*Bb
!-----------------------------------------
;
          mult[i]=1,np : Bb([.i],j)= f_d(deta_tef[i])(/ff(j));
C ===== FAMILY-1 == CL  =====
      DO J=1,MP1
      BB1(1,J)=(( (F_DELTA(JO1+2,JO1+J))*(PI_PRANDTL))-( (0.)))
      BB1(2,J)=(((-(0.))+( (F_DELTA(JO1+1,JO1+J))*(PI_RAYLEIGH_RATIO)))+
     *(-(0.)))
```

```
      ENDDO
C ===== FAMILY-2 == DWN-CL =====
      DO J=1,MP2
      BB2(1,J)=(F_DELTA(JOU+ 1,JO2+J))
      BB2(2,J)=(0.)
      ENDDO
C ===== FAMILY-2 == NDE-CL =====
      DO J=1,MP2
      DO INODE=1,NNODE(2)
      BB2(K2I_1(INODE),J)=((F_DELTA(JO2+K2J_1(INODE),JO2+J))-( (0.)))
      BB2(K2I_2(INODE),J)=((F_DELTA(JO2+K2J_2(INODE),JO2+J))-( (0.)))
      ENDDO
      ENDDO
C ===== FAMILY-2 == UP-CL  =====
      DO J=1,MP2
      BB2(IOFUP2+1,J)=(0.)
      BB2(IOFUP2+2,J)=(0.)
      ENDDO
C ===== FAMILY-U == CL  =====
      DO J=1,MPU
      BBU(1,J)=(0.)
      ENDDO
! Il faut egalement deriver / ix_Phi
! ----------------------------------
C ===== FAMILY-1 == CL  =====
      DO J=1,MPU
      BB1(1,MP1+J)=(( (F_DELTA(JO1+2,JOU+J))*(PI_PRANDTL))-( (0.)))
      BB1(2,MP1+J)=(((-(0.))+( (F_DELTA(JO1+1,JOU+J))*(PI_RAYLEIGH_RATIO
     *)))+(-(0.)))
      ENDDO
C ===== FAMILY-2 == DWN-CL =====
      DO J=1,MPU
      BB2(1,MP2+J)=(F_DELTA(JOU+ 1,JOU+J))
      BB2(2,MP2+J)=(0.)
      ENDDO
C ===== FAMILY-2 == NDE-CL  =====
      DO J=1,MPU
      DO INODE=1,NNODE(2)
      BB2(K2I_1(INODE),MP2+J)=((F_DELTA(JO2+K2J_1(INODE),JOU+J))-( (0.))
     *)
      BB2(K2I_2(INODE),MP2+J)=((F_DELTA(JO2+K2J_2(INODE),JOU+J))-( (0.))
     *)
      ENDO
      ENDDO
C ===== FAMILY-2 == UP-CL  =====
      DO J=1,MPU
      BB2(IOFUP2+1,MP2+J)=(0.)
      BB2(IOFUP2+2,MP2+J)=(0.)
      ENDDO
C ===== FAMILY-U == CL  =====
      DO J=1,MPU
      BBU(1,MPU+J)=(0.)
```

```
ENDDO
```

Tout ça nous coûte un peu moins de 400 lignes de macros, sachant qu'on a déjà pas mal optimisé leur découpage.

Les probes. On doit pouvoir les interlacer avec les IX_Tr, et il faut un stack à par : ce sera le stack [4], qui sert aux Do_Loop, ce qui entraine qu'il est interdit de créer une set_Probes<> à l'intérieur d'une do_loop. Et qui, donc, se rapproche du fonctionnement des IX_Tr, sans nécessité de maillage **me semble-t-il ?**. On peut donc le générer a partir de la suite mult[]ff(.) classique. De même que pour la 102, on choisit par principe minimum de ne pas leur associer de matrice D entre eux, jusqu'à nécessité avérée. Les probes (Mu) sont déjà montrées en exercice dans la matrice D des extraits précédents.

Ainsi, on peut résumer le principe général de création du code Fortran :

- pour une matrice, les mult génèrent autant de lignes que de multiplicité en ff(.) et en eta(.), c'est-à-dire le nombre des équations, qui diffère du nombre de variables dans les cas maillés ;
- Mortran reconnait chaque ligne – genre D(1,.) – et si c'est maillé, fait précéder la dérivée d'une ouverture de boucle pour la premiére équation d'un bloc de mailles, et la ferme pour la dernière ;
- à chaque fois, on pointe sur le bon sous-vecteur avec un zéro-pointeur (io ou jo) suffixé par l'ordinal de la famille, de même pour les ofsets de ligne de matrice (dans le cas Up, on ajoute j ou iofup[Fm]) ;
- la seule différence concernant les IX_Tr et les probes, c'est que les boucles courent sur toutes les composantes des vecteurs (1,MP ou 1,NP) ;
- ce qui fait qu'on ne calcule pas les extensions **do J=1,MPU** pour ces objets ;

Il suffira que le $DoDimEtaPhi renseigne correctement les bornes des mult[] pour que le tout soit automatique. **Dans la mesure où le code Fortran généré est efficace, on retient cette solution pour la 102.**

2.4 Principe des macros SET_Eta_Phi

On l'illustre sur les transferts internes non-maillés, mais on donne également les DWN et UP qui utilisent les mêmes macros et permettent par comparaison de reconnaitre l'intéret de la forme choisie des macros.

Les principes suivent le traitement des blocs Mortran habituels,c'est-à-dire qu'à l'ouverture, on fixe des paramètres définis par le type de bloc, et on génère les macros de fermeture - c'est l'objet des P du stack [1]. Les instructions write donneront la trace en sortie de listing des céations, et l'appel a PROCHLP – qui n'est pas encore correct – permettra de générer d'une part le Model.hlp, d'autre part de conserver les noms de variables pour informer les lignes et colonnes des matrices et vecteurs, mais aussi de générer une en-tête aux fichiers .data.

On compte l'ordinal de l'équation d'un transfert dans le stack [6] ; et comme on devra traiter un début de boucle, la première FUN= est différente des suivantes.

Le stack [8] compte les objets dans le bloc, puisque'à chaque variable correspondra une ligne des matrices, possiblement décalée par jofup[7], le stack [7] étant le cardinal de la famille du bloc.

```
&

';IN_PHI<'='#LP#S1#L #S8;WRITE(*,*);
WRITE(*,*)'' ------ Informing on Phi def.  in family :'',#U7#S7#C,'' -----'';
;$CRPHI$;
 &'',##$FUN:##=##;''=''$DOP1(##1,##2,##3,##U6##S6##C,##U7##S7##C,1,Tr )
 &''''####K,####$FUN:####=####;''''
 &'''',####$FUN:####=####;''''='''''####U6####A1####S6####U8####A1####S8
$DOP(####1,####2,####3,####U6####S6####C,##U7##S7##C,####U8####S8####C)''''''';'
! -------------------------------
! macro set for dwn and up objects
! -------------------------------
&';DWN_PHI<'='#LP#S1#L #S8;WRITE(*,*);
WRITE(*,*)'' ------ Informing on Dwn-Phi   in family :'',#U7#S7#C,'' -----'';
$CRPHI$;
! le f_set  du nom de la fonction
 &'',##$FUN:##=##;''=''$DOP1(##1,##2,##3,##U6##S6##C,##U7##S7##C,1,Dwn-Tr)
 &''''####K,####$FUN:####=####;''''
 &'''',####$FUN:####=####;''''='''''####U6####A1####S6####U8####A1####S8
$DOP(####1,####2,####3,####U6####S6####C,##U7##S7##C,####U8####S8####C)''''''';'
!------------ UP set ---------------
&';UP_PHI<'='#LP#S1#L #S8;WRITE(*,*);
WRITE(*,*)'' ------ Informing on  Up-Phi   in family :'',#U7#S7#C,'' -----'';
$CRPHI$;
! le f_set  du nom de la fonction
 &'',##$FUN:##=##;''=
''$DOP1(##1,##2,##3,##U6##S6##C,##U7##S7##C,jofup##C+1,Up-Tr)
 &''''####K,####$FUN:####=####;''''
 &'''',####$FUN:####=####;''''='''''####U6####A1####S6####U8####A1####S8
$DOP(####1,####2,####3,####U6####S6####C,##U7##S7##C,
jofup##C+####U8####S8####C)''''''';'
! ------------------------------------------------------------
&'$CRPHI$'=';
WRITE(*,*)''   Var-name,           Function-name,       index in ff vector'';#N
 &''VAR:##,''=''OVARHLP='''''##1'''';,##1$''
 &''$P$''=''##U8
;WRITE(*,*)'''' ------------------------------------------------------------'''';
 &''''####K$P$'''' &''''####KVAR:####,'''' &''''####K,####$FUN:####=####;''''
 &''''####RCT(##U6##C,J)=F_D(####)(/ETA(J))''''='''';ENDDO;''''
 &''''####RD(##C,J)=F_D(####)(/FF(J))''''='''';ENDDO;''''
 &''''####RD(##C,J)=F_D(####)(/FF(JOU+J))''''='''';ENDDO;''''##A1##S6'''
```

On constate que pour un Up-Tr, on a un simplement l'offset **jofup[7]** en plus du pointeur courant du stack [8], qui devra être calculé une fois pour toute en fonction du nombre de mailles, lui même rangé dans le tableau M_Node([Fm]).

On rappelle que ceci est effectué grâce à une série de macros dédiées lors d'un premier écrêmage du ZINIT (transparent bien sûr à l'utilisateur), c'est la fameuse $DoDimEtaPhi déjà utilisée dans la 102.

Finalement, on informe deux macros qui vont générer le calcul des dérivées du bloc, une première d'ouverture, une deuxième de ligne courante. Pour la fermeture des blocs, on ajoute ENDDO quand nécessaire.

Les macros ';#* comment ; permettent de suivre la structure dans le code Fortran.

```
! -------- contenu des derniers stacks : - ---------------------
!            #4:[6],#5:[7],#6:0_ptr, #7:Cmt(Up...)
&'$DOP1(#,#,#,#,#,#,#)'='F_SET0 #1=ff(jo(#5)+#6);f_set #2=#3;
f_set Phi_tef(#4)=#3;
 &''FF(#4)=PHI_TEF(#4)''=''FF(jo(#5)+#6)=PHI_TEF(#4)''
 &''DOT_PHI(#4)=F_D(##)(/time)''=''DOT_PHI(jo(#5)+#6)=F_D(##1)(/time)''
 &''CT(#4,J)=F_D(##)(/ETA(J))''=';####* ===== Family-#5 === #7 =====;
DO J=1,NO#5;CT#5(#6,J)=F_D(##1)(/ETA(io(#5)+J))''
 &''CT(#4,NOT+##)=F_D(''=';CT#5(1,NOT+##1)=F_D(''
 &''D(#4,J)=F_D(##)(/FF(J))''=';####* ===== Family-#5 == #7 =====;
DO J=1,MO#5;D#5(#6,J)=F_D(##1)(/FF(jo(#5)+J))''
 &''D(#4,J)=F_D(##)(/FF(JOU+J))''=';#### ===== Family-#5 == #7 =====;
DO J=1,MOU;D#5(#6,MO#5+J)=F_D(##1)(/FF(joU+J))''
WRITE(*,''(2A24,I3)'')''#1'',''#2'',#6;IODXHLP=#6;
OFNAHLP=''#2'';OFUNHLP=''#3'';OFMYHLP=FmNam(#5);
CALL PRHLP2(2,IODXHLP,jo(#5),0,0,OVARHLP,OFNAHLP,OFUNHLP,#5,OFMYHLP);'
&'$DOP(#,#,#,#,#,#)'='F_SET0 #1=ff(jo(#5)+#6);f_set #2=#3;
f_set Phi_tef(#4)=#3;
 &''FF(#4)=PHI_TEF(#4)''=''FF(jo(#5)+#6)=PHI_TEF(#4)''
 &''DOT_PHI(#4)=F_D(##)(/time)''=''DOT_PHI(jo(#5)+#6)=F_D(##1)(/time)''
 &''CT(#4,J)=F_D(##)(/ETA(J))''=''CT#5(#6,J)=F_D(##1)(/ETA(io(#5)+J))''
 &''CT(#4,NOT+##)=F_D(''=''CT#5(#6,NOT+##1)=F_D(''
 &''D(#4,J)=F_D(##)(/FF(J))''=''D#5(#6,J)=F_D(##1)(/FF(jo(#5)+J))''
 &''D(#4,J)=F_D(##)(/FF(JOU+J))''=''D#5(#6,MO#5+J)=F_D(##1)(/FF(joU+J))''
WRITE(*,''(2A24,I3)'')''#1'',''#2'',#6;IODXHLP=#6;
OFNAHLP=''#2'';OFUNHLP=''#3'';OFMYHLP=FmNam(#5);
CALL PRHLP2(2,IODXHLP,jo(#5),0,0,OVARHLP,OFNAHLP,OFUNHLP,#5,OFMYHLP);'
! ------------------------------------------------------------
```

Deux dérivations supplémentaires à celles du TEF sont reconnues :

- celle par rapport aux paramètres pour la Ct(.,.), reconnue par le fait qu'elles sont rangées en extension dans Ct(.,NP[Fm]+.) ;
- celle par rapport aux transferts externes (/FF(JOU+J)), car ces composantes sont rangées en fin du tableau des FF(.), et ainsi en non-continuité avec FF(jo[Fm]+1 :jo[Fm]+MP[Fm]), le sous-bloc vecteur de la famille [Fm] (stack [7]). Le zéro-pointeur de ces ff(.)s externes étant joU (famille U comme Univers) ;
- à part ce dernier cas, les composantes du transfert interne ne doivent être sensibles qu'aux composantes internes de la famille, soit eta(io[Fm]+1 :io[Fm]+NP[Fm]) pour la Ct, ff(jo[Fm]+1 :jo[Fm]+MP[Fm]) pour la D, la famille du bloc ayant NP[Fm] composantes d'état et MP[Fm] composantes de transfert ;

Le cas des blocs de mailles est aussi simple, si ce n'est qu'il doit en plus gérer le passage de l'indice de maille Var(imode) à l'ordinal de la composante dans le vecteur-tableau ff(.), ce qui se fait par construction des fonctions implicites du genre k2j_[8] de la 102 en mode Grid1D. On ne le décrira pas ici.

Pour finir, indiquons que l'ouverture de la famille Univers initialise le stack [6] premettant de lister les équations de transferts que va déclarer l'utilisateur – ainsi bien sûr que le comptage des familles dans le stack [7].

Tout ceci est semblable à la déclaration des cellules, avec en plus absence de particularité pour l'éventuelle cellule directement sous U.

2.5 Dimensionement et indices

On a à la base deux multiplicités **nomlt** et **momlt**, à savoir les nombres d'équations d'états et de transferts. Ces nombres diffèrent de **np** et **mp**, les nombres respectifs de variables, du fait des maillages éventuels. Le nombre des familles internes est nof-1, la famille englobante U prenant le numéro **nof**.

On repère dans les deux tableaux-vecteurs **eta(.)** et **ff(.)** les sous-vecteurs asociés à chaque famille par zéro-pointeur : **jo[Fm]** et **io[Fm]**, où on indique par [Fm] l'ordinal de la famille, de 1 à **nof)**, la famille U se distingant par le symbole **u**.

Dans chaque famille, on garde le nombre des variables **NP[Fm]** et **MP[Fm]**, avec MPU comme nombre des transferts externes. On garde également pour les familles maillées un **iofdwn[Fm]** ou **jofdwn[Fm]**, soit le nombre des down-objects, qui jouera le rôle de zéro-pointeur pour les mailles, et nomlt[Fm], la multiplicité sur chaque maille. Le nombre des mailles est n ou monde(k).

Tous ces pointeurs suffixés sont mis en équivalence avec les tableaux dimensionnés **jo[i] :=jo(i)**, etc.

On aura alors les relations :

- io(1) = jo(1) = 0 ;
- io(k) = io(k-1) + NP(k-1) et jo(k) = jo(k-1) + MP(k-1),
- reste le passage var(inode) à la composante. On utilise des fonctions implicites comme k2i_3(inode)=k2i_3+iofdwn2+nomlt2*(inode-1)+3, dans laquelle le 2 est le numéro de la famille, le 3 indique qu'il s'agit de la troisième équation de chaque maille ;
- on pointe sur les Up-objects par i ou jofup(k) = jo(k)+MP(k) ;
- avec MP(k) = +nonde(k)*momlt(k) ;
- idem pour cellule maillée ;
- MP = somme des MP(k), de 1 à nof+1, NP de même ;
- le cas du transfert externe est plus compliqué du fait qu'il permet une alternance de parties maillées ou non. Une solution possible est d'avoir un offset qui s'incrémente par bloc d'équations jox([9]) – le stack [9] étant réservé à ce transfer – c'est-à-dire qu'à la fin d'un bloc de déclaration de mailles, il s'incrémente de la différence entre la multiplicité et le nombre de variables.

Pour des raisons de limitation du nombre des stacks disponibles, on choisit une limitation à la création libre de transferts externes ... **non, plus le 26 décembre.**

Les **transferts externes** devant rester très souples, on doit pouvoir alterner des créations de composantes maillées ou non sans contrainte. Le pointage sur les composantes est alors plus lourd :

- comme pour les IN, on a un zéro-pointeur de vecteur **JOU** ;
- à chaque equation est associé un un-pointeur **JOX([9])** ;
- celui-ci, qui est calculé au cours de la lecture du ZINIT standard, s'incrémente d'une unité dans les cas SET,DWN et UP, et du nombre de mailles dans le bloc IX_[#]PHI¡ en cours de traitement ;
- on a donc un pointage vecteur par JOU+JOX([9]) si non maillé, JOU+KUJ_[9](inode) si maillé, ce choix étant effectué en fonction du numéro de ligne d'une matrice ;
- alors, on a par exemple si la septième équation est dans une grille KUJ_7(inode)==JOX(7)+(inode-1)*MOMLU1, ce dernier "1" marquant la multiplicité du bloc maillé local (repéré en réalité par l'indice [9] de la première équation du bloc) ;

Ce système de ointage et comptage permet de créer ad-libidum une alternance de blocs maillés avec des nombres de mailles différents. On peut donc tout se permettre avec ces U-Tr, mais il faudra être d'autant plus rigoureux sur la connectique que l'on aura utilisé cette extrême souplesse.

Le pointage sur une ligne de Jacobienne est semblable mais sans zéro-pointeur io ou

jo, puisque les matrices sont séparées. Mais on les rassemble dans un common par type : **common/AMx/A1,A2,A3,...,A ;**, et il faut donc pouvoir pointer sur $n^{ième}$. On présente donc ici les dimensions et un-pointeurs en désignant la troisième famille pour alléger :

- A3(NP3,NP3+LPar) ; A3(1,1) := AOMx(koa(3)) ;
- koa(k)=koa(k-1)+NP(k-1)*(NP(k-1)+LPar) ;
- B3(NP3,MP3+MPU) ; B3(1,1) := BOMx(kob(3)) ;
- kob(k)=kob(k-1)+NP(k-1)*(MP(k-1)+MPU) ;
- Ct3(MP3,NP3+LPar) ; Ct3(1,1) := COMx(koc(3)) ;
- koct(k)=koct(k-1)+MP(k-1)*(NP(k-1)+LPar) ;
- D3(MP3,MP3+MPU) ; D3(1,1) := DOMx(kod(3)) ;
- kod(k)=kod(k-1)+MP(k-1)*(MP(k-1)+MPU) ;
- pour l'externe, Ct(MPU+MObs,MPU+LPar) ;
- et D(MPU+MObs,MPU) ;

avec **LPar** le nombre des paramètres libres et **MObs** le nombre des probes, l'ensemble des paramètres de base devant être déterminés par $DoDimEtaPhi et les moins basiques calculés en tête du ZINIT.

2.6 Génération de listes

Pour générer les listes de dimensionement, common, etc des vecteurs et matrices, on retiend[1] que **nof** est le nombre de familles internes, leur ordinal marquant les objets associés, et la marque par "U" est celle de la cellule (éventuelle) et le transfert directement situés sous la famille Univers. On a ainsi à générer, pour la trice **A** par exemple :

```
real Bb1(NO1,MO1+MOU),Bb2(NO2,MO2+MOU),Bb3(NO3,MO3+MOU),BbU(NOU,MO
```

Ceci se fait par une macro **m_liste** :

```
real m_list [I](=1,nof):A[.I](NO[.I],NO[.I]+LPar),AU(NOU,NOU+Lpar);
common/MAx/m_list [I](=1,nof):A[.I],AU;
```

Cette macro reconnaait une première partie à développer, et une terminaison connue à conserver. La voici :

```
&'$L(#$[#,#]$##:#)'='#U5$KL#C$#A1#S5
 &''$KL#C$''='' &''''####K$L(#1#4####[.#2]####$''''
 &''''####K$L(#1#4####(####[.#2]####)$''''''
 &''$L(#1#4##[.#2]##$''='')$L(#1#4##1#C##2$''
 &''$L(#1#4##(##[.#2]##)$''='')$L(#1#4##1(##2#C##3)$''
$L(#1#4#5$[#2,#C]$#4:#5)'
;
! liste (,) avec fin connue
&'$MLST[#,#](=#):#,#;'='#M#1#S5 &''$KL#C$''='''
 &''$L(##$[#2,#3]$,:#4)''='' &''''####K$L(####$[#2,#3]$,:#4)''''
 &''''####K$KL#3$'''' &''''$FML$;''''='''##1,#5;''''##U5''
 &''$L(##[.#2]##$''='')$L(##1#C##2$''
 &''$L(##(##[.#2]##)$''='')$L(##1(##2#C##3)$''
$L(#4$[#2,#C]$,:#4)$FML$;'
;
&'m_list[#](=#,#):'='$MLST[#2,#1](=#3):'
```

La forme est d'abord transformée pour mettre en premier l'argument de début de liste, seule forme permettant de tranférer cette valeur dans un stack, par #M#1#S5 comme

[1] le 27 décembre 2007.

ici, qui est mise en stack [5]. La macro $MLST prépare comme d'hab la terminaison c'est-à-dire une $L avec valeurs finales, puis les macros $L de premier remplacement de [.I] par la première valeur de I.

La macro $L standard traite les valeurs suivantes jusqu'à la dernière. Alors, la $L de fin est reconnue, c'est elle qui écrit la liste (dans ##1) et la fin connue (#5). Chaque nouvelle incrémentation du stack tue les macros de remplacement générées pour la valeur précédente : c'est fait par les $KL[5], préparées au vol, mais utilisées quand le stack est incrémenté, donc en différé.

On a aussi à compter des dimension globales connaissant les locales :

```
parameter(not=no1+no2+no3+no4)
parameter(mot=mo1+mo2+mo3+mo4)
```

(en rapellant que les NO4 et MO4 sont egaux aux NOU et MOU).

On utilise une ad_liste très semblable à la m_liste. Décrivons la comme étant plus simple :

```
! add list
&'$ALST[#,#](=#):#);'='#M#1#S5 &''$KL#C$''='''
 &''$L(##$[#2,#3]$+:#4)''='' &''''####K$L(####$[#2,#3]$+:#4)''''
 &''''####K$KL#3$'''' &''''$FML$;''''='''##1);''''##U5''
 &''$L(##[.#2]##$''='')$L(##1#C##2$''
 &''$L(##(##[.#2]##)$''='')$L(##1(##2#C##3)$''
$L(#4$[#2,#C]$+:#4)$FML$;'
;
&'ad_list[#](=#,#):'='$ALST[#2,#1](=#3):'
```

On a la mème transformation pour placer la valeur initiale en premier argument (#1). Donnons le démarrage en trace de Mortran :

```
  0         parameter (not=no1+ad_list [I](=2,nof+1):no[.I]);
 MATCHED TEXT   'ad_list[I](=2,nof+1):'
 PATTERN        'aD_LIST[#](=#,NOF+1):'
 PARAMETER 1    'I'
 PARAMETER 2    '2'
 REPLACED TEXT  'ad_list[I](=2,4):'
 MATCHED TEXT   'ad_list[I](=2,4):'
 PATTERN        'aD_LIST[#](=#,#):'
 PARAMETER 1    'I'
 PARAMETER 2    '2'
 PARAMETER 3    '4'
 REPLACED TEXT  '$ALST[2,I](=4):'
 MATCHED TEXT   '$ALST[2,I](=4):no[.I]);'
 PATTERN        '$ALST[#,#](=#):#);'
 PARAMETER 1    '2'
 PARAMETER 2    'I'
 PARAMETER 3    '4'
 PARAMETER 4    'no[.I]'
 REPLACED TEXT  ' &'$KL2$'='' &'$L(#$[I,4]$+:no[.I])'=' &''##K$L(##$[I,4]$+:no[.I])''
                  &''##K$KL4$'' &''$FML$;''=''#1);''#U5'
                  &'$L(#[.I]#$'='$L(#12#2$' &'$L(#(#[.I]#)$'='$L(#1(#22#3)$'
                  $L(no[.I]$[I,2]$+:no[.I])$FML$;'
```

Il a fallut d'abord explciter **nof+1**, ce qui est préparé par DODIMETAPHI. Paramètrfe initial en premier argument (dît PARAMETER dans Mortran), cf $ALST. Le vrai

démarrage de liste est dans le dernier REPLACED TEXT. On prépare d'abord la tueuse de macro de remplacement $KL2 pour plus tard.

On prépare aussi la macro de terminaison de liste avec la valeur finale (ici 4) : $L(#$[I,4]$+ :no[.I]), qui lance la dernière tueuse de remplaceuse déjà prète, puis qui la tue (##K$KL4$). Une macro reconnaissant 'FML;" est lancée qui écrit le résultat (dans #1). Il faut enfin vider le stack de travail (#U5).

La macro de fin étant préparée, on doit faire les macros de premier remplacement $L des [.I] par "2" cette initiale valeur, les remplacements suivant étant pris en charge par la même $L standard que pour la m_liste.

2.6.1 Autres listes (15 Jan 08)

On a rencontré un problème avec la m_list, qui provient de la souplesse du TEF. Les règles sont les suivantes :

- il y a forcément une cellule dans une famille intérieure ;
- mais pas forcément de transferts internes ;
- Il y a forcément un transfer sour U ;
- mais une cellule ou pas sous U ;
- il peut n'y avoir aucune famille interne ;

La conséquence est ainsi qu' à une famille est associé ou non une matrice **Ct**, qu'il n'est pas question de dimensionner nulle, car interdit par Fortran. On ne peut donc pas systématiser les listes en suite de nombres entiers consécutifs, il faut construire les liste. On procède dans dodimetaphi à la construction de ces liste et on les utilise de la manière suivante dans **principal**.

Il y a une liste des transferts internes, une de cellules, qui celle-ci est consécutive mais peut s'arrêter avant **U**.

On a ainsi à présent les déclarations symboliques, ici pour **Bb** et **Ct** : :

```
real m_list[I]:Bb[.](NO[.],MO[.]+MOU);
common/BbMx/ m_list[I]:Bb[.];
equivalence u_list[I]:(BobMx,Bb[.]);

real m_list[J]:Ct[.](MO[.],NO[.]),CtU(MOU+Mobs,NOT+LPar);
common/CtMx/ m_list[J]:Ct[.],CtU;
equivalence u_list[J]:(CoMx,Ct[.]);
```

qui vont par exemple générer

```
      real Bb1(NO1,MO1+MOU)
      common/BbMx/Bb1
      equivalence (BobMx,Bb1)
      real CtU(MOU+Mobs,NOT+LPar)
      common/CtMx/CtU
      equivalence (CoMx,CtU)
```

On constate qu'on n'a pas de in-Tr, donc pas Ct ni D, et $NO1 = 0$. Les m_list sont marquées pat [I] ou [J] pour cellules et transferts.

```
! macros d'arret
&',$L[.]:$#;'='#1;#N'
&':#,$L'='#1#N,$L'
! macros courantes
&',$L[.]:#$[#]$'='
 &'',##[.]##,$L[.]:$''='',##1#2##2,$L[.]:$''
```

```
 &'',##(##[.]##),$L[.]:$''='',##1(##2#2##3),$L[.]:$''
,#1,$L[.]:$'
&',$L[.]:#$[#,#]$'='
 &'',##[.]##,$L[.]:''='',##1#2##2,$L[.]:''
 &'',##(##[.]##),$L[.]:''='',##1(##2#2##3),$L[.]:''
,#1,$L[.]:#1$[#3]$'
&',$L[.]:#$[U,#]$'=',$L[.]:#1$[#3,U]$'
! premieres macro
&'$L[.]:#$[#]$'='
 &'':##[.]##,$L[.]:''='':##1#2##2,$L[.]:''
 &'':##(##[.]##),$L[.]:''='':##1(##2#2##3),$L[.]:''
:#1,$L[.]:$'
&'$L[.]:#$[#,#]$'='
 &'':##[.]##,$L[.]:''='':##1#2##2,$L[.]:''
 &'':##(##[.]##),$L[.]:''='':##1(##2#2##3),$L[.]:''
:#1,$L[.]:#1$[#3]$'
! mise en format commande
&'$[#]$L[##]:#;'='$L[.]:#3$[#1]$;'   "a priori, que pour [L]"
&'$[,#]$L[##]:#;'='$L[.]:#3$[#1]$;'
&'$[,#]$L[##]:#,#;'='$L[.]:#3$[#1]$,#4;'
&'$[,U]$L[J]:#;'=';'       "valable pour J liste, pas L ni I"
&'$[,U]$L[J]:#,#;'='#2;'
&'$[]$L[J]:#,#;'='#2;'
;
! uniques macros de u_liste
&'$[#].$L[.]:#;'=' &''(##[.]##);''=''(##1#1##2);''#2;' "[L] seul ?"
&'$[,#].$L[.]:#;'=' &''(##[.]##);''=''(##1#1##2);''#2;'
&'$[,#,#].$L[.]:#;'=' &''(##[.]##);''=''(##1#1##2);''#3;'
&'$[].$L[.]:#;'=' &''(##[.]##);''=''(##1U##2);''#1;'  "***"
;
!**&'$[,#,#].$L[.]:#,#;'='$L[.]:#2$[#1]$,#4;'
! macros de commande
&'m_list[##]:#;'='$LST[#1]$L[#1]:#2;'
&'u_list[##]:#;'='$LST[#1].$L[.]:#2;'
```

On commente (toujours) des dernières macros en remontant (comme le fait Mortran). On place en tête une ,liste $LST[I] ou [J] - l'argument ## en pattern signifie "un seul caractère. Ces liste ayant été construites dans dodimetaphi, les LST sont remplacées par une vrai liste de forme $[,2,3,4,U]. Il va donc suffire de construire la liste des symboles comprenant un [.] en construisant, pour chaque item de la liste, les macros substituant l'item (comme 3 en deuxième) aux symboles [.]. On recommence avec une liste plus courte. On doit savoir arrêter une liste, son ultime forme ne comprenant plus qu'une seule virgule (cf macros courantes). On enlève alors les fins de lignes inutiles (macros d'arrêt). La présence de U (mis en parameter Fortran égal à $(nof + 1)$) est traité à part dans le cas [J], du fait du dimensionnement particulier des matrices du transfert externe.

On remarque la u_liste qui ne doit prendre que le premier item d'une liste, pour l'équivalence avec la forme de tableau de rangement des matrices :
equivalence (D0Mx(1),D2(1)).

2.7 Calculs de sensibilité

On garde la structure de la 102 en étendant les vecteurs d'états et tranferts par les colonnes des matrices de sensibilités-propagateurs, mais il pourra s'avèrer plus judicieux de les segmenter, à voir à la programmation.

Ceci dit, ce calcul :

```
!  Source pour propag. de transfert marche : gam_t  -> + |BbDm1>_k dt
!              de type g/1-g
! ---------------------------------------------------------------------
       if jo.eq.1
       < call omatmlt(n,m,m,Bb,np,un_moins_D,mp,bobuf,np);
         call oscamat(dt,n,m,bobuf,np,bobuf,np);
         Do ko=1,nxp
         < if ko_P_typ(ko) .eq. 4
           < call omatadd(n,1,bobuf(1,ko_p_kvar(ko)),np,gam_Phit(1,ko),np
                                                 ,gam_Phit(1,ko),np);
           >;
         >;
       >;
     >;"End nxp"
```

n'avait l'avantage que d'exploiter le fait que (I-D)m1 est connu - ce qui n'est pas posible pour la 200. On résoudra cette fois-ci $Bx = (I - D)$ en sa seule k-ième colonne. Ou rassembler les colonnes k et proceder à une élimination.

3 Les variables et symboles, rangements

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

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

nom	dim 1	dim 2	*signification*	data file
eta(1 :np1)	np1	1	vecteur d'état	res1.data
dot_eta(id.)	np1	1	vel d'état	no data
dneta(id.)	np1	1	avance d'état	dres.data
Phi_t(1 :np1,∗)	np1	nxp+1	matrice des eta-sensibilités aux variables	sens1.data
dPhi_t(id.)	np1	nxp+1	matrice des eta-sensibilités aux variables	no data
Phi_p(id.)	np1	lpar+1	matrice des eta-sensibilités aux paramètres	sensp1.data
dPhi_p(id.)	np1	lpar+1	matrice des eta-sensibilités aux paramètres	no data
ff(1 :mp1)	mp1	1	vecteur de transfert	tr1.data
ff(mp+1)	mobs	1	vecteur des observables (sous U)	obs.data
Psi_t(1 :mp1,∗)	mp1	nxp+1	matrice des ff-sensibilités aux variables	sigma1.data
Psi_p(id.)	mp1	lp+1	matrice des ff-sensibilités aux paramètres	sigmap1.data
Pso_t	mobs	nxp+1	matrice des mu-sensibilités aux variables	sigmo.data
Pso_p	mobs	lp+1	matrice des mu-sensibilités aux paramètres	sigmop.data

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

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

(où NP et MP sont les sommes des dimensions internes augmentées de celles sous U).

Un ensemble de symboles Mortran sont dans les instructions de calcul des matrices et vecteurs de **principal**, qui seront remplacées par des expressions ou des symboles différents selon les directives générées à partir du **ZINIT**. Les voici, quasi identiques à la 102 :

nom	*signification*	origine du remplacement
deta_tef[i]	$g_i(\eta, \varphi)$	fun : de set_eta < ... >;
(/oPi[j])	composante de ϖ	free_parameters : liste
phi_tef[i]	$f_i(\eta, \varphi)$	set_phi < ...,fun :... ; ...>;
(/phi_tef(JOU+[.i])	derivee / transfert externe	fun : de set_obj < ..., > ;

on remarque que les symbole nde_eta_fun et nde_ff_fun de la 102 ont disparus.

Les matrices du TEF :

Jacobiennes	sous-matrices	*signification*	TEF-mx
H1	H1(1 :np1, 1 :np1)	$\partial_{\eta_1} g_1(\eta_1, \varphi_1)$	$A1 = I - \frac{\delta t}{2} H1$
	H1(1 :np1, np1+1 :np1+lpar)	$\partial_{\varpi} g_1(\eta_1, \varphi_1, \varpi)$	id
Bb1	Bb1(1 :np1, 1 :mp1)	$\partial_{\varphi_1} g_1(\eta_1, \varphi_1)$	$B1 = -\frac{\delta t}{2} Bb1$
Bbu_1	Bbu_1(1 :np1, 1 :mp1)	$\partial_{\varphi_1} g_1(\eta_1, \varphi_1)$	$Bu_1 = -\frac{\delta t}{2} Bbu_1$
Ct1	Ct1(1 :mp1, 1 :np1)	$\partial_{\eta_1} f_1(\eta_1, \varphi_1)$	Ct1
	Ct1(1 :mp1, np1+1 :np1+lpar)	$\partial_{\varpi_1} f(\eta_1, \varphi_1, \varpi)$	Ct1
	Ct1(mp1+1 :mp1+mobs, 1 :mp1)	$c1_2 = \partial_{\eta_1} f_\mu(\eta_1, \varphi_1)$	Ct1
	Ct1(mp1+1 :mp1+mobs, np1+1 :np1+lpar)	$c1_1 = \partial_{\varpi} f_\mu(\eta_1, \varphi_1, \varpi)$	Ct1
D1	D1(1 :mp1, 1 :mp1)	$\partial_{\varphi_1} f_1(\eta_1, \varphi_1)$	D1
	D1(mp1+1 :mp1+mobs, 1 :mp1)	$\partial_{\varphi_1} f_\mu(\eta_1, \varphi_1, \varpi)$	D1

On conserve donc la structure de la 102 dans chaque famille, contrairement à la présentation (plus claire pour les calculs) du grand tableau matriciel précédent : les bandeaux de couplage désignés sont en réalité distribués en extension des matrices de base des familles internes.

4 Principe des calculs

On décrit la base du calcul des trajectoires (et les extensions sensibilité[1])

- **Montée** : **call MOVUP**
- boucle sur les familles
  - élimination des cellules **EOLES**
  - élimination des transferts **EOLEF**
- Sous **U** :
  - élimination de la cellule éventuelle **EOLES** ;
  - résolution **EOLMAX** ;
  - avance état cellule sous U ;
- **Descente** : **call MOVDWN**
  - mise à jour des transferts internes **ODLF** $\rightarrow$ **ODWNPHI**
  - avance cellules **ODETA** $\rightarrow$ **ODWNETA**
- avances des observables et des sensibilités associées ;

Les routines MOVUP et MOVDWN voient donc toute la structure de l'arbre à deux niveaux. Celle-ci est explicitée par des pointeurs et dimensions sur les tableaux de vecteurs-matrices du TEF.

Prenant l'exemple des matrices A_i, A_u, rangées dans un common ; deux tableaux NOP(I), KNOP(I) suffisent à itérer sur l'indice I des familles internes[2], le dernier I=NOF correspondant à U. On aura alors un algorithme du genre :

```
      SUBROUTINE MOVUP(NOP,KNOP,MOP,KMOP,A,...,ETA,...FF,...)
       DIMENSION NOP(*),KNOP(*)
       DIMENSION A(*) ...
! -- Calcul des pointeurs globaux
     Do kof=2,NOF+1
     < kAof(kof) = kAof(kof-1)+
! -- Elimination des cellules
      Do kof=1,NOF+1
      < Call EOLES(NOP(kof),A(ktof(kof),...,Eta(itof),...)
```

[1] où on s'est amusé à reprendre les symboles de ZOOM, mais l'amusement pourra être utile à qui s'efforcerait de joindre les deux codes.

[2] attention, il apparait que ces notations changent suivant le chapitre.

```
      >;
! -- Elimination des transferts internes
      Do kof=1,NOF
      < Call EOLEF
      >;
```

Pour les rangements de matrices, on aurait :
- EOLES fait le calcul de $[A_1^{-1}B_1]$ et $A_1^{-1}\Gamma_1$, nécessaires à la descente, que l'on range à la suite de B_1 et de Gamma_1,x ;
- EOLEF calcule $[I - D_1 + C_1^\dagger A_1^{-1}B_1]$ et $[D_1^u + C_1^u A_1^{-1}B_1]$. On calculera donc en une fois les deux termes en $C^\dagger$, et on rangera la matrice des couplages internes à la suite de la D_1 étendue de sa partie D_1^u ; idem pour les vecteurs $\delta\varphi_{ins}\Omega_1 - C_1^\dagger A_1^{-1}\Gamma_1$ qui sont gardés à la suite du tableau dFi,x — ou alors déjà dans dFi,x ? sauf qu'on imprime le dFi_ins des gains ...
- EOLMAX fait essentiellement les mêmes calculs que **oker** mais à partir des matrices couplage ;
- pour le calcul du gain par Borel, il suffira de le faire à ce niveau (oui ?) ;
- descente par ODWNF ; on range les $\delta\varphi$ et extensions comme d'hab ;
- ODWNETA calcule $\delta\eta$;

5 Les algorithmes

5.1 Calcul de la trajectoire

Sans changement par vrapport à la 102, l'enchaînement des calculs reste :
- le ZINIT : description du modèle et initialisations ;
- (avec un premnier scanning de ZINIT élaborant les dimensionements) ;
- **début de la boucle temporelle** $\longleftarrow \bullet$;
- La **D-LOOP** : résolution du système implicite $\varphi = f(\eta, \varphi)$;
- calcul de la matrice Jacobienne D par dérivation symbolique ;
- calcul des autres matrices Jacobiennes H, Bb et Cr par dérivation symbolique ;
- extensions : calcul des Ψ (Phi_t et Sens_p) ;
- initialisation éventuelle particulière de Φ ;
- sortie des variables (qui sont en phase après la D-loop) ;
- calculs diagnostics (matrice d'avance d'état A^{spha1}, fonction de coût J) et sauvegardes pour l'adjoint ;
- test de linéarité de l'avance de φ ($f(t+\delta t) \leftrightarrow f(t) + d\varphi$) ;
- calcul des vecteurs et matrices du TEF (intervention de δt) ;
- sur demande,(boucle sur δt si balayage de Borel$\rightarrow$ gain, et dans ce cas, retour su δt de l'avance temporelle ;
- préparation analogue pour le calcul des extensions ;
- **appel à oker pour la résolution** ;
- séquence ZSTEP(pour gestion de δt et calculs particuliers) ;
- écriture fichier de $\delta t, \delta\eta$;
- avance η et $t, \delta t$ (et extensions) ;
- passage par la séquence ZSTEER ;
- **fin du pas de temps, boucle** $\bullet \nearrow$;
- fermeture fichier et arrêt.

On pourra envisager d'y ajouter des calculs formatés comme les Lyapunov et Floquet, comme les points de lecture de données qui ne sont pas ici explicités et ne devraient pas

[1] On peut déjà remarquer que cette matrice ne pourra être découpée en blocs, et devra être de diimension NP, ce qui me semble-t-il ajoute une difficulté pour l'avance du filtre de Kalman,
Pat, qu'en (quand) penses-t-tu ?.

différer de la 102.

du ZINIT au calcul des Jacobiennes

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

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

```
!%%%%%%%%%%%%%%%%%%%%
! Cell definition
!%%%%%%%%%%%%%%%%%%%%
Under Universe
< under Family-1
  < set_eta
    < var: eta_courant_L,
      fun: deta_conv = pi_prandtl*ff_T_czcx
                                      - pi_prandtl*eta_courant_L;
      var: eta_T_czcx,
      fun: deta_energy_zx = - ff_Psi_Tsz + pi_Rayleigh_ratio*ff_courant_L
                                                    - eta_T_czcx;
      var: eta_T_sz,
      fun: deta_energy_z = ff_Psi_Tczcx - pi_wave_nb_ratio*eta_T_sz;
    >;
  >;
>;
```

On a défini trois variables d'état, chacune d'elles associée à un "modèle", c'est-à-dire un équation d'évolution $\partial_t \eta = g(\eta, \varphi)$. Ces équations font usage de variables de transfert (ff_xxx) déclarées par ailleurs. L'expression $g()$ sera dérivée par rapport aux composantes de η (matrice H) et φ (matrice B_b). Le calcul de ces dérivées est écrit en utilisant les symboles standards de variables d'état et sa formule correspondante :

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

En premier lieu, le symbole **nomlt** de la **mult** est remplacée par "3", grâce à des macros travaillant sur le code de **dodimetaphi** (généré dans la 200). Ensuite, la macro **mult** génère trois lignes avec incrémentation de [i] de 1 à 3 (La **set_eta** a mis un compteur à la valeur 1, et).

La deuxième ligne, par exemple, est alors

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

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

- de remplacer "H(2," par "H1(2,", où on a indicé le symbole de matrice par l'ordinal de la famille – qui ici est unique ;
- et d'associer **deta_tef(2))** à la formule déclarée en deuxième position :

[1] en progression.

```
H1(2,j) = f_d( - ff_Psi_Tsz
                  + pi_Rayleigh_ratio*ff_courant_L
                                        - eta_T_czcx
                                                    )(/eta(j));
```

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

```
H1(2,j) = f_d( -ff(2) +pi_Rayleigh_ratio*ff(3) -eta(2))(/eta(j));
```

– il ne reste alors plus qu'à effectuer la dérivation **(/eta(j))**.

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

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

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

On conçoit alors d'emblée l'importance de la définition d'un langage, puisque non seulement il permet de séparer la formulation algorithmique du code des symboles du calcul, mais de plus il permet de générer du code Fortran supplémentaire pour vérifier, conserver et transmettre de l'information tout au long de la chaîne de calcul, dans les print_out et dans les fichiers résultats. Au moment où ces lignes sont écrites, seul Mortran offre d'aussi vertigineuses potentialités[1] **mais ceci devrait interpeller la génération montante de férus d'informatique pour, soit trouver (lambda-calcul ?) une meilleure solution, soit se convaincre de la nécessité d'une ré-écrirure de Mortran étendant ses potentialités**[2]**.**

La D-Loop

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

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

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

avec $D = \partial_\varphi f(\eta, \varphi)|_{\varphi_0}$. La variable vectorielle φ_0 est dans le tableau **phiz** (routine **newt**). Il faut donc calculer la matrice D en dérivant **f(**φ_z**)** en fonction de **phiz**, d'où les macros

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

[2] On a déjà dans le cahier des charges de pouvoir enchaîner plusieurs passage de Mortran avec des paquets de macros, d'étendre le nombre de stacks, d'étendre les possibiltés de calcul dans le registre arithmétique, mais maussi peut-être de prévoir un ou plusieurs stacks de chaîne de caractères

MTN transformant les formulations de **Phi_tef(ff)** en **Phi_tef(phiz)**, qui sont annulées en sortie de la D-Loop ("retour à ff").

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

La 200 suit le même algorithme mais avec son unique niveau d'emboîtement. Dans la 102, toute la dépendance des tranferts entre eux était déterminée par cette seule équation $\varphi = f(\eta, \varphi)$, où n'intervient que la matrice D, η étant fixé. Dans la 200, on a autant de matrices que de famille, le transfert externe leur ajoutant un badeau-colonne et un bandeau-ligne de couplage. On a ainsi un seul niveau d'emboîtement similaire au système complet mais après élimination des cellules. On apellera alors la mème routine **EOLMAX** avec des matrices-couplage restreintes aux D_i..

Fin du pas de temps et sorties des variables

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

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

Tous les résultas sont écris dans des fichiers **.data** assortis éventuellement d'un suffixe différent (**osuffix**) comme décrit dans le Manuel pratique. Dans la 200, le problème se pose de la génération de fichiers .data par famille ou non. **On a donc suppression de l'option Grid1D de la 102.**

transformation du cas du Grid 1D

Contrairement à la 102, on a vu que c'est la reconnaisance de "D(3," par exemple qui permet de tenir compte d'une équation définie dans un bloc maillé, Up ou Dwn. C'est donc à partir de la même séquence de **mult** que tout est fait, à ceci près qu'on remplace les indices courants de ligne par une fonction de passage de (inode) à la ligne du tableau, ou via un offset.

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

Les matrices Jacobiennes A, Bb, Ct sont calculées de manière semblable à D. Le calcul des vecteurs **ff(.)** et **dot_eta(.)** aussi, mais sans indexation par l'ordinal de famille.

Dans la mesure où ces calculs de Jacobiennes interviennent dans l'adjoint, leur code reste regroupé pour chacune dans une séquence.

Enfin pour finir avec les matrices du TEF, les matrices $A = I - \frac{\delta t}{2} H$ et $B = -\frac{\delta t}{2} Bb$ sont effectuées, non plus dans le corps de **principal**, mais du fait de la séparation en blocs de matrice dans une subroutine **call H2A(nop,HoMx,nop,AoMx,nop)**, et **Bb2B(nop,mop,MPu+Mobs,BobMx,nop,BoMx,nop)**.

5.2 La routine oKer, solveur du système

Ayant en argument d'entrée les quatre matrices d'avance $A, B, C^\dagger, D$ du TEF et les vecteurs connus Γ, Ω, le sous-programme **oker** effectue la résolution du système (**??**) en deux temps : élimination des cellules menant au système dit des "couplages" entre les

transferts, résolution des transferts, et enfin "descente" pour résolution des états. On a en sortie les variations $\delta\eta$ et $d\phi$.

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

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

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

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

Les calculs et tableaux sont organisés comme suit dans la 102, avec commentaires pour la 200 :

- "vecteurs" connus rassemblés dans la matrice **bggx** dans l'ordre **bw, gamma, gammx** ; de cette manière, la "montée" déterminant le système des couplages fournit à la fois la matrice $A^{-1}B$ et les variations découplées $A^{-1}\Gamma, \Gamma_x$ après l'appel à un premier **sgesv** (avec le flag LaPack **infoa**). On a ainsi **mp+1+nwp** colonnes de résultat. *200→ idem mais traitement par bande-lignes de chaque famille. On crée ainsi un common rassemblant les Aom_i, avec NP_i nombre de lignes (comme AoMx), mais avec nwp+1 colonnes en plus des MP_i (MP_1+LPar pour AoMx) : common/AmB/AmBx1(NP1,MP1+nwp+1),AmBx2(NP2,MP2,...,AmBxU(NPU,MPU+nwp+1), calculé par EOLES avec tableau équivalent AomBx(.) et tableau de pointeurs sur les AmBx_i ;*
- il reste à multiplier par $C^\dagger$ pour obtenir la matrice de couplage d'une part, les évolutions insensibles $(C^\dagger A^{-1}\Gamma, \Gamma_x)$ d'autre part. *200→ EOLEF effectue les calculs et rangement de CoMx(k)AomBx(k), rangés dans Couplage(k), de même nombre de lignes que les D_i avec MD_i colonnes plus (nwp+1). On a donc un autre common /couplages/Coupx(k), en équivalence avec Coupx1(MP1,MP1+nwp+1),...,CoupU(MPU+Mobs,MPU+nwp+1) ;*

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

   call omatmlt(mp,np,mp+1+nwp,ct,mp+mobs,bggx,np,ctbggx,mp);
200-> avec Do-Loop sur      CoMx(k),mop(k),AomBx(k),nop(k),coupx(k),mop(k))
++++++++++++++++++++ et à part :  CoMx(nof+1),  etc

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

   CALL omatadd(mp,mp,c_unit,mp,ctbggx,mp,unplusctamb,mp);
   CALL omatsub(mp,mp,unplusctamb,mp,d,ldd,couplage,mp);
   !                                        ========
200-> ibidem
++++++++++++
   if Zprint
   <     TITLE='>ker : Matrice de couplage';
         CALL printmat(title,couplage,mp,mp,mp,mp,2,2);
   >;
```

```
200-> ibidem
++++++++++++
   !  *********** calcul de dphi_ins  ***********
   !    dphi_ins = Omega - (-Ct)A-1.Gamma

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

– on est à présent "en haut de navette" dans le système de couplage, avec un "vecteur connu" **dphixx** de **1+nwp** colonnes. *200→ Il reste encore un niveau d'emboîtement, soit élimination des transferts internes. On applique alors EOLMAX qu travaille sur les coupx(.). Et comme il faut conserver les matrices et vecteurs pour la redescente de navette, on a un dernier common de coumDx(l) pointés par les mêmes pointeurs que pour coupx;*
Ici intervient l'option Borel-sweep, avec création du vecteur $\mid 1_i >$ rangé en **1+nwp+1**
200→ On a ici une difficulté, car soit on retiend l'option ZOOM consistant à ne calculer que des gains de Borel de transferts externes, soit on offre plus de facilitè, et on initialise le dphixx(i_gain,nwp+2)=one avant l'élimination précédente. Le principe de souplesse nous fait ici choisir cette dernière option, ce qui supprime le IF<>ELSE<> suivant.

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

200→ EOLMAX effetue cette dernière résolution.

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

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

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

```
   ! Calcul avance etat
   DO I=1,np
   < dneta(i)=agamdt(i)-ambdphixx(i,1);
```

```
          if (ZPRINT)
          write(*,9001)i,agamdt(i),ambdphixx(i,1),dneta(i),eta(i),
                                                  EtaNam(i)(1:lenocc(EtaNam(i)));
        >;
        9001;format(2x,I3,4(1Pe12.4),1x,A);
```

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

- *200→ ibidem avec ce niveau supplémentaire qui consiste à calculer les $\delta\varphi_k$, avec la routine ODWNPHIà l'aide des coupmx(k) montants, puis les $\delta\eta_i$ avec ODWNETA à l'aide des AomBx(i) montants*;

En sortie de oKer

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

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

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

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

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

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

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

```
      >;
    >;
>;
```

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

Déjà dans la 102, on a créé un tableau **off_norm(.)** *pour pondérer ces écarts correctement si l'utilisateur veut bien l'informer :*

```
if (istep.gt.1)
< res=0.; <io=1,m; res = res +(ffl(io)-ff(io))**2/off_norm(io); >;
```

se pose alors le problème du défaut, qui est l'unité dans la 102. Peut-être peut-on souhaiter reprendre le test précédent par défaut, mais alors les autres usages (éventuels) de cette norme, de même que celle sur les états (oeta_norm(.)) sont á gérer. On pense en effet à ces normes pour régulariser certains calculs de sensibilité, Floquet et autres. **à voir.**

Trajectoire en arrière

L'intégration à la TEF du système adjoint donne :

$$\begin{bmatrix} -I - \frac{dt}{2}H^\dagger & -\frac{dt}{2}C \\ -B_b^\dagger & I - D^\dagger \end{bmatrix} \begin{bmatrix} \delta v \\ \delta w \end{bmatrix} = \begin{bmatrix} \Gamma_v \\ \Omega_w \end{bmatrix} + \begin{bmatrix} l_\eta \delta t \\ l_\varphi \end{bmatrix} \tag{3}$$

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

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

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

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

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

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

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

! Fonction de cout  (dt<0) par trapèzes
  coutstep = - (coutlbef+cout_l)*0.5*dt;
```

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

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

+SEQ,trans_abcd. et calculs vecteurs                        (D)

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

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

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

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

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

! ------------------------------------------------------------
!  ********** calcul du w debut de istep avec D et B bef **
```

[1] extension possible mais non encore envisagée.

```
!             ( I-Dt) w = Bt v + l_y
! ----------------------------[sauf a l'entree]-----------
```

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

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

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

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

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

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

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

200→ *on applique le même principe, qui fait qu'après transposition des matrices, et le changement dt en −dt la résolution est standard. Il pourra ètre plus économe de partir du tronc commun des Jacobiennes et d'effectuer le passage −dt et transposition par blocs au moment des opérations pour conserver les pointeurs de trajectoire directe.* **à voir**

6 DODIMETAPHI

Le principe reste le même que dans la 102 : on utilise un premier écrêmage de ZINIT pour déterminer les paramètres de dimensionnement des matrices et vecteurs, en supprimant toutes les instructions de la séquence. Il faut de plus déterminer la forme des diverses fonctions intrinsèques nécessaires au pointage des objets maillés. Enfin, on y prépare des macros tuant dans le code principal les blocs logiques de type ";IF NXP.gt.O¡ ... ¿" pour un paramètre nul (ici NXP).

Partons du résultat attendu des différenrs comptages dont on dispose à la fermeture du bloc **Under Universe** :

```
&';UNDER UNIVERSE<'='#LU#S1#L/31#S4#S5#S6#S9#L #S7;
 &''$U$''='';####* ============ End of Fm Universe ============;
;####* ...... Total Nber of Fm= ##U7##C;parameter(nof=##C);
parameter(NO##C=NOU);parameter(MO##C=MOU);
 &''''m_list[####](=####,nof):''''='''m_list[####1](=####2,####C):''''
 &''''ad_list[####](=####,nof):''''='''ad_list[####1](=####2,####C):''''
;####* ...... Total Nbr of transfer Eq= ##U6##S6##C, and Cls = ##U5##C;
parameter (nomult=##C,momult=##U6##C);
;####* ...... ----- Number of IX-Tr Eq= ##U9##C;
;####* ...... ----- Number of Probes= ##U4##C;parameter (mobs=##C);'';'
```

La fermeture du bloc, marquée oar U, utilise le nombre des familles, comptées par le stack 7 ([7]), les multiplicités d'équations de cellules (nomlt=[5]) et de transferts (momlt=[6]). La famille U étant par convention d'ordinal NOF, on équivalence NO[nof] et NOU, de même pour les transferts. On remarque ce choix final de systématiser l'utilisation de la lettre "O" en position deux dans les noms de variables (NP→NO).

Enfin, on a généré les macros de liste (ad et m_list) qui auront l'effet de compléter les déclarations symboliques dépendant de NOF :

```
parameter (nopt=no1+ad_list [I](=2,nof):no[.I]);
parameter (mopt=mo1+ad_list [I](=2,nof):mo[.I]);
real m_list [I](=1,nof):H[.I](NO[.I],NO[.I]+LPar);
common/HMx/m_list [I](=1,nof):H[.I];
```

qui donnera :

```
      parameter(nopt=no1+no2+no3)
      parameter(mopt=mo1+mo2+mo3)
      real H1(NO1,NO1+LPar),H2(NO2,NO2+LPar),H3(NO3,NO3+LPar)
      common/HMx/H1,H2,H3
```

Les instructions Mortran du genre ";##* ..." génèrent une carte commentaire, pour débogage, si bien que la fermeture U génère :

```
C ============ End of Fm Universe ============
C ...... Total Nber of Fm= 3
      parameter(nof=3)
      parameter(NO3=NOU)
C ...... Total Nbr of transfer Eq= 19, and Cls = 9
      parameter (nomult=9,momult=19)
C ...... ----- Number of IX-Tr Eq= 10
C ...... ----- Number of Probes= 3
      parameter (mobs=3)
```

A l'ouverture, on initialise les stacks, soit avec la valeur nulle (#L[CTR^]), soit l'unité ($L)[1]. Le stack [4] compte les probes - qui elles aussi se déclarent *ad libidum* mais sans maillage. Le stack [9] est réservé aux U-Tr.

La déclaration des blocs SET doit initialiser [8] qui compte les équations du bloc.

```
&';IN_PHI<'='#LP#S1#L'0'#S8;$CRPHI$;'
&';DWN_PHI<'='#LP#S1#L'0'#S8;$CRDPHI$;'
&';UP_PHI<'='#LP#S1#L'0'#S8;$CRUPHI$;'
! ------------------------------------------------------------
&'$CRPHI$'='; &''VAR:##,##;''=''##U6##A1##S6##U8##A1##S8;''
 &''$P$''='' &''''####K$P$'''' &''''####KVAR:####,####;''''
;parameter(MO##U7##S7##C=##U8##C);
;####* --- Tr mlt= ##C, total= ##U6##S6##C;''';
&'$CRDPHI$'='; &''VAR:##,##;''=''##U6##A1##S6##U8##A1##S8;''
 &''$P$''='' &''''####K$P$'''' &''''####KVAR:####,####;''''
;parameter(MODWN##U7##S7##C=##U8##C);
;####* --- DwnTr mlt= ##C, total= ##U6##S6##C;''';
&'$CRUPHI$'='; &''VAR:##,##;''=''##U6##A1##S6##U8##A1##S8;''
 &''$P$''='' &''''####K$P$'''' &''''####KVAR:####,####;''''
```

[1] le code Mortran a été modifié judicieusement par bibi pour accepter cette valeur initiale nulle, ce qui fait ganger des dizaines de lignes de macros (le 21 décembre 2007). Dans l'exemple, #L/31 est réalité le caractère de contrôle : ascii(RS) ; on le note '0' dans la suite.

```
;parameter(MO##U7##S7##C=MODWN##C+MOGRID##C+##U8##C);
;####* --- UpTr mlt= ##C, total= ##U6##S6##C;'''';
! ------------------------------------------------------------
&';IN_[#]PHI<'='#LP#S1#L'0'#S8;$CRNPHI$[#1];'
! ------------------------------------------------------------
&'$CRNPHI$[#]'=' &''VAR:##(##),##;''='''##U6##A1##S6##U8##A1##S8;
;####*    function k##U7##S7##CJ_##U8##S8##C(inode)=MODWN##U7##S7##C+(inode-1)*
MOMLT##C+##U8##S8##C;''
 &''$P$''='' &''''####K$P$'''' &''''####KVAR:####(####),####;''''
;parameter(MOGRID##U7##S7##C=##U8##S8##C*#1,MOMLT##U7##S7##C=##U8##C,
JOFUP##U7##S7##C=MODWN##C+MOGRID##C);
####* ---- Grid Tr mult= ##C, total= ##U6##S6##C, M_nodes: #1;'''
```

Chaque CRPHI, pour chacune des lignes (VAR= , FUN=), en effectue un comptage local en incrémentant [8] et global [6]. A la fermeture des blocs, P peut générer le PARAMETER de dimensionnement.du vecteur correspondant au bloc.

Il faut bien sùr calculer le passage de la multiplicité au cardinal pour les mailles. On utilise alors MOMLT[fm] le nombre d'équation par maille, MODWN[fm] le nombre de DWN composantes, ainsi que MOGRID[fm] le nombre de composantes maillées. Un commentaire explicite la fonction de maille en attendant la déclaration. On obtient typiquement :

```
      parameter(MODWN2=1)
C --- DwnTr mlt= 1, total= 10
C    FUNCTION k2J_1(inode)=MODWN2+(inode-1)*MOMLT2+1
C    FUNCTION k2J_2(inode)=MODWN2+(inode-1)*MOMLT2+2
      parameter(MOGRID2=2*3,MOMLT2=2,JOFUP2=MODWN2+MOGRID2)
C ---- Grid Tr mult= 2, total= 12, M_nodes: 3
      parameter(MO2=MODWN2+MOGRID2+1)
```

où JOFUP2 permet de pointer les éventuelles composantes UP du transfert. MO2 servira a dimensionner les matrices de cette 2ème famille et à calculer la dimension totale du vecteur des transferts (qui pourra se faire dans l'ignorance des parties maillées) ; de plus, on s'en servira pour calculer (en FTN dans ZINIT passe 2) les zéro-pointeurs de vecteurs.

7 FLOQUET

Pour des raisons diverses et pas toujours très claires, on cherche pour une série de calculs utilisants les Jacobiennes du SLT à les rassembler dans l'algorithme d'avance, en évitant un passage systématique en complexes. Le calcul de vecteurs de Floquet fait partie de ce lot, puisqu'il prend des valeurs initiales parmi celles des V.P. de $\Phi(T,0)$ qui peuvent être complexes. On remarque que vu d'une couche informatique, le rangement en mémoire du nombre complexe se fait sur deux "mémoires réelles" consécutives. Hors, pour les matrices réelles, on a nécessairement des paires complexes conjuguées, l'idée est de séparer les calculs qui traitent chaque composante indépendamment et ceux qui les croisent. Ceci minimise la séquence de calculs en complexe.

Pour Floquet,on adopte la procédure décrite dans les "Propagateurs" et la formule :

$$\delta\Psi^F = \delta\Psi^r \exp(-\Lambda\tau) - \Psi^r \exp(-\Lambda\frac{\tau}{2})(I + \frac{(\Lambda\tau)^2}{24})\Lambda\tau \qquad (4)$$

dans laquelle Ψ^r comporte la succesion des colonnes du Floquet complexe, et où $\delta\Psi^r$ est avancé par **oker**. Ainsi, les colonnes sont remises à leur conformité complexes par les seules

corrections (diagonales) données par (4). On aura avantage à les expliciter en fonction de $\lambda = \lambda_r + i\lambda_i$, en calculant successivement :

$$\begin{aligned} \Re: \quad & \alpha = \lambda_r \left[1 + (\lambda_r^2 - \lambda_i^2)\tfrac{\tau^2}{24}\right] - \lambda_i(\lambda_r \lambda_i \tfrac{\tau^2}{12}) \\ \Im: \quad & \beta = \lambda_i \left[1 + (\lambda_r^2 - \lambda_i^2)\tfrac{\tau^2}{24}\right] + \lambda_r(\lambda_r \lambda_i \tfrac{\tau^2}{12}) \end{aligned} \tag{5}$$

multiplier par $e^{-\lambda_r \frac{\tau}{2}}[\cos(\lambda_r \frac{\tau}{2}) - i \sin(\lambda_r \frac{\tau}{2})]$

$$\begin{aligned} \Re: \quad & \gamma = e^{-\lambda_r \frac{\tau}{2}}[\alpha \cos(\lambda_r \tfrac{\tau}{2}) - \beta \sin(\lambda_r \tfrac{\tau}{2})] \\ \Im: \quad & \zeta = e^{-\lambda_r \frac{\tau}{2}}[\beta \cos(\lambda_r \tfrac{\tau}{2}) + \alpha \sin(\lambda_r \tfrac{\tau}{2})] \end{aligned} \tag{6}$$

puis de réarranger les colonnes $\mid r > + i \mid j > \rightarrow$:

$$\begin{aligned} \mid r > \rightarrow \quad & \gamma \mid r > -\zeta \mid j > \\ \mid j > \rightarrow \quad & \gamma \mid j > +\zeta \mid r > \end{aligned} \tag{7}$$

8 Eléments Propres

L'idéal serait d'emboîter, et on s'y essaie ici, si possible. Les v.p.sont solutions de :

$$Det[A - \lambda I + B(I - I)^{-1} C^\dagger] = 0 \tag{8}$$

que l'on récrit successivement

$$\begin{aligned} Det. &= Det[(A - \lambda I)(I + (A - \lambda I)^{-1} B (I - D)^{-1} C^\dagger] \\ &= Det[(A - \lambda I)] Det[I + C^\dagger (A - \lambda I)^{-1} B (I - D)^{-1}] \\ &= Det[(A - \lambda I)] Det[I - D + C^\dagger (A - \lambda I)^{-1} B][(I - D)^{-1}] \\ &= \frac{Det[(A - \lambda I)]}{Det[(I - D)]} Det[I - D + C^\dagger (A - \lambda I)^{-1} B] \end{aligned}$$

où on a utilisé le théorème des permutations. On ne peut s'empêcher de sentir comme un frémissement d'emboîtement possible ... ?

9 Construction des routines des navettes

9.1 EOLEF

Cette routine intervient dans la D_loop, dans la navette d'avance temporelle, mais aussi dans divers algorithmes liés aux calcul des sensibilité[1]. On a à résoudre par complémentation de Schur l'équation :$(I-D)X = Y$, Y donné dans le mx-vecteur **Omega**.

La structure matricielle de $I - D$ correspondant à MK2 est ici réduite à un seul niveau d'emboîtement :

```
  I-D1   0    |-D1u   avec [D1,D1u] ensemble in Domx(kod(i))
   0   I-D2   |-D2u
   0     0 ...|...
 ------------------
-[Du1, Du2,...|I-Du
          \_ in Domx(kou+kodu(i))
```

De plus, dans le cas de la navette complète, après élimination des cellules, les blocs D_{ij} ont déjà pu hériter des $C_i^\dagger A_i^{-1} B_j$, auxquel il faut à présent ajouter $I - D_i$ ou $-D_{ij}$. Voici l'entête du code et son appel :

```
call EoleF(nof,mObs,mot,noxx,noxx,oMgmx,time,istep,iPiv,Zprint,Zdebug);
;
subroutine EoleF(nof,mObs,mot,noxx,ndim,oMgmx,time,istep,iPiv,Zprint,Zdebug);
implicit none;
+SEQ,General_dim.
+SEQ,TR_Mx.
integer nof,ndim,u,k;
real oMgmx(mot+mObs,0:*);
! time values
real time; integer istep;
logical Zprint,Zdebug;
character*32 title;
! LaPack
integer iPiv(mot+mObs),infoa;
;
! Initialisations
joU=jo(nof+1); moU=mo(nof+1); moI= mot-moU; u=nof+1;
;
```

Les arguments sont en premier des paramètres de dimension, avec nof le nombre des familles internes, mObs celui des probes, mot le nonbre total des composantes de transfert, noxx=nox+lpar+1 le nombre des vecteurs connus (formant Y) rangés dans la matrice-vecteur **oMgmx**, et enfin **ndim** le nombre de colonnes de Y. Le reste des arguments a des fins de printout.

Ces entrées sont complémentées par des common ; un premier, qui est général, fournit l'accès aux noms d'objets pouor du printout, mais on a encore les tableaux **io** et **jo** des pointeurs sur les sous-vecteurs d'état et de transfert. Enfin des notations utiles particluières pour la famille **U**, d'ordinal **nof+1** :

```
+SEQ,General_dim.
integer not,mot,mobs,lpar,maxstep,nox,noxx,noy,noz;
```

[1]Le nom EOLEF est repris de ZOOM, construit sue élimination et F comme familles, la lettre "o" en position 1 ou 2 réservant ce nom au noyeau de ZOOM.

```
character*24 FmNam(1),EtaNam(1), PhiNam(1),ParNam(1);
common/prhlp/FmNam;
common/prhlpa/EtaNam;
common/prhlpb/PhiNam;
common/prhlpc/ParNam;
integer io(1),jo(1),joU;
common/ioptr/io;common/joptr/jo;
integer no(1),mo(1),moU;
common/noptr/no;common/moptr/mo;
```

Ensuite, une séquence regroupant les obejts liés aux transferts, matrices $C^\dagger$ et D, avec leurs pointeurs. Enfin, les matrices du calcul montant et descendant dont on va voir l'usage cî-après.

```
+SEQ,TR_Mx.
! Jacobiennes
! -----------
real CoMx(1),DoMx(1);
common/CtMx/CoMx;common/DMx/DoMx;
! Matrices de descente et de couplage
! -----------------------------------
real ComCpMx(1),CoupUMx(1);
common/cmcpMx/ComCpMx;common/CoupU/CoupUmx;
! pointeurs de matrices et vecteurs
! ---------------------------------
integer moI;
! Jacobiennes et Couplages
integer koc(1),kod(1),kop(1),kocu(1),kodu(1);
common/cptr/koc;common/dptr/kod;common/cmcp/kop;
common/cu/kocu;common/du/kodu;
```

La première partie du code consiste à compléter par les $I - D_i$ ou $-D_ij$ ce qui a éventuellement déjà été rempli par l'élimination des cellules - ou mis à zéro antérieurement :

```
! complete CoupU,x with OmegU and CoupU = -Diu,I-Duu           AAAAA
! ------------------------------------------------
call omatsub(moU+mObs,mot,CoupUmx,moU+mObs,Domx(kod(u)),moU+mObs,
                                                    CoupUmx,moU+mObs);
call omatadU(moU,moU,CoupUmx(kodu(u)),moU+mObs,
                                                CoupUmx(kodu(u)),moU+mObs);
call omatadd(moU+mObs,ndim,Omgmx(joU+1,0),moU+mObs,
                                 CoupUmx(kodu(u)+(moU+mObs)*moU),moU+mObs,
                                 CoupUmx(kodu(u)+(moU+mObs)*moU),moU+mObs);
;
! (IX, fm ->IX)
! -------------
do k=1,nof
<if (mo(k).gt.0)                                                BBBBB
 <
! ---------------------------------------------
! now computes Coup_i,u = I - D_i [+ CtAmBGmx,i]deja fait
! ---------------------------------------------
   call omatadU(mo(k),mo(k),Comcpmx(kop(k)),mo(k),Comcpmx(kop(k)),mo(k));
   call omatsub(mo(k),mo(k)+moU,Comcpmx(kop(k)),mo(k),
```

```
                                  domx(kod(k)),mo(k),Comcpmx(kop(k)),mo(k));
;
! and extension Coup_i,u = Omegx_i + (CtAmGmx,i)                CCCCC
! ----------------------------------------
   call omatadd(mo(k),ndim,Comcpmx(kop(k)+mo(k)*(mo(k)+moU)),mo(k),
               Omgmx(jo(k)+1,0),mo(k),Comcpmx(kop(k)+mo(k)*(mo(k)+moU)),mo(k));
;
 >;
;
! en entree, on a Cp_i,u,x,
! en sortie Cpim1Cpiu,x dans vecteur de Comcpmx
! ----------------------------------------                    DDDDD
;
  call sgesv("in:" mo(k),moU+ndim,Comcpmx(kop(k)),mo(k),iPiv,
                        "in/out" Comcpmx(kop(k)+mo(k)*mo(k)),mo(k),infoa);
>;
```

Les matrices **Comcp_i** concernent une bande pour chaque famille. On range à gauche les $I - Di$ suivis des $-D_i u$ et du vecteur Y pour son sous-bloc de la famille. La bande correspondant à la famille **U** est dans le tableau **CoupUmx**, avec cette fois à gauche la liste des $-D_{ui}$, puis la partie diagonale $I - D_u$ suivie d'un bloc de Y. vecteur. Ainisi, on va calculer $[I - D_u + \sum_i D_{ui}(I - Di)^{-1}[D_{iu}, \Omega_i]]$ qui sera rangé dans la parie diagonale de **CoupUmx** suivie du vecteur.

- A : on soustrait $D_{u,1}, D_{u,2}, ..., D_u$ à **CoupUmx** en une seule fois (**mot**), et on ajoute I à sa diagonale. **kodu(u)** pointe cette diagonale. On copie également Ω_u sur **ndim** colonnes ;
- B : idem pour les bandes **Comcpmx** ;
- C : on place les mx-vecteurs Ω_i à droite des bandes, en ajoutant à ce qu'il peut déjà y être ($C^\dagger A^{-1}\Gamma$ de la famille) ;
- D : on calcule $(I - D_i)^{-1}[D_{iu}, \Omega_i$, dont le résultat est à droite du bloc diagonal de **Comcpmx** pointé par **kop(k)+mo(k)*mo(k)**. Ces blocs seront nécessaires à la descente ;

Il reste à effectuer l'élimination des transferts internes :

```
;
! IX,fm,IX
! ---------
do k=1,nof
<if (mo(k).gt.0)
  call omatmltsb(moU+mObs,mo(k),moU+ndim,CoupUmx(kodu(k)),moU+mObs,
                                  ComCpmx(kop(k)+mo(k)*mo(k)),mo(k),
                                            CoupUmx(kodu(u)),moU+mObs);
>;
return;
end;
```

en utilisant la routine **matmltsb**, qui soustrait chaque terme de la somme $\sum_i D_{ui}(I - D_i)^{-1}D_{iu}$ à **CoupUmx_diag**, en traitant en même temps la partie vecteur. En effet en entrée, les $-D_{iu}$ sont dans chaque **CoupUmx(kodu(k), Comcpmx(kop(k)+mo(k)*mo(k))** pointe sur les $(I - D_i)^{-1}[D_{iu}, \Omega_i]$; il faut soustraire parce qu'on $-D_{ui}$, mais on y était obligés par souci de rentrer les bons signes des $C^\dagger A^{-1}B$ pour ne pas multiplier les risques d'écarts au formalisme.

Comme on a pris l'exemple de la navette d'avance d'état, on traite tout le vecteur, qui est composé à la fois des tranferts et des extensions : sensibilité aux paramètres et les

différents Ψ, avec à terme des parties de calcul des Floquet. D'où la valeur **noxx** donnée à l'argument d'entrée **ndim**.

On peut encore remarquer le traitement des probes (**moU+mObs**) en nombre de lignes du tranfert externe, ayant en effet prévu qu'ils puissent avoir un couplage par D.

Les autres utilisations comprennent :

- la **D_loop**, où on aura **ndim =1**, puisque la seule équation implicite non-linéaire concerne le vecteur des transferts. Dans ce cas, le vecteur Ω a été initialisé avec le résidu **ff-phiz** :

```
! raz Comcpmx and CoupUmx
! ----------------------
 call veczero(Comcpmx,noptot); call veczero(CoupUmx,(moU+mObs)*(mot+1));
! residue ff-phiz in Omega
! ------------------------
call omatcopy(mot,1,ff(1),mot+mObs,                  Omgmx(1,0),mot+mObs);
call omatsub (mot,1,Omgmx(1,0),mot+mObs,phiz(1),mot,Omgmx(1,0),mot+mObs);
call EoleF(nof,mObs,mot,noxx,1, Omgmx,time,istep,iPiv,Zprint,Zdebug);
```

- il faut calculer $\Psi_{t,p}$ après la **D_loop**, par $(I-D)\Psi_{t,p} = C^\dagger \Phi_{t,p} + [sources\ \Phi, C^\dagger_\pi]$, que l'on range dans $\Omega_{,x}$;
- enfin, la source pour effet-en-retour se calcule à partir de $B_b(I-D)^{-1}\Phi_t$. On pense sélectionner chaque colonne Φ_i nécessaire, effectuer la navette pour le calcul de $(I-D)^{-1}\Phi_i$ avant de multiplier à gauche par B_b ; on pourra éventuellement stocker les colonnes à calculer dans le mx-vecteur Ω et n'avoir qu'une seule navette de genre **D_loop** avec **ndim** donnant le nombre des colonnes-source ;

On a de plus évidemment les extensions du genre Floquet pour après la **D_loop**, qui se trouveront en extension de Φ, et les calculs de l'adjoint, qui *a priori* ne concerneront que l'avance rétrograde, comme dans la **102**.

9.2 EOLES

Cette routine intervient seulement dans l'avance temporelle (ou retro)[1]. On a à former par complémentation de Schur l'équation :$(I-D+C^\dagger A^{-1}B)X = Y$, et $Y = \omega + C^\dagger A^{-1}\Gamma$.

La structure matricielle dans MK2 est ici, avec deux familles par exemple :

```
A1  0  |  B1    0      | B1u   avec [B1,B1u] contigues dans Bomx(kob(k))
                                      suivies de Gam,x
0   A2 |  0     B2     | B2u
-----------------------------

Ct1 0  | I-D1   0      |-D1u   avec [D1,D1u] ensemble in Domx(kod(i))
                                      suivies de Omeg,x
0   Ct2|  0    I-D2    |-D2u
-----------------------------
Cu1,Cu2,-[Du1, Du2,...|I-Du
                \_ in Domx(kou+kodu(i))
```

L'élimination des cellules passe par le calcul de matrices de descente dans chaque famille-k interne :$[A_k^{-1}B_k, A_k^{-1}\Gamma_k]$. On range et conserve ces matrices dans les tableaux **AomBGmx**, organisés comme suit :

1. à l'initialisation, on range par famille $[A_k, B_k, B_{ku}, \Gamma_k]$, avec en tête la matrice A_k ;
2. on calcule $A_k^{-1}[B_k, B_{ku}, \Gamma_k]$, la partie en-téte ne sera plus utilisée ;

[1] Le nom EOLES est repris de ZOOM, construit sur élimination et S comme State.

3. il reste à effectuer les produits par $C_k^\dagger$ pour initialiser les blocs dîts "de couplage" : **ComCpMx(kop(k))** ;
4. le cas de la longue $C_U^\dagger$ est enfin traité avec initialisation de **CoupU** ;

Voici l'en-tête du code et son appel :

```
call EoleS(nof,mObs,not,mot,noxx,noxx,GoMx,time,istep,iPiv,Zprint,Zdebug);
;
subroutine EoleS(nof,mObs,not,mot,noxx,ndim,Gomx,time,istep,iPiv,Zprint,Zdebug);
implicit none;
+SEQ,General_dim,Cl_mx,Tr_mx.
integer nof,ndim,u,k,noU;
real Gomx(not,0:*);
real time; integer istep;
logical Zprint,Zdebug;
character*32 title;
integer iPiv(not),infoa;
! Initialisations
noU=no(nof+1); joU=jo(nof+1); moU=mo(nof+1); moI= mot-moU; u=nof+1;
! en entree, on a B et Gamx separes,
! en sortie AmB,Gx sont concatenes dans AomBGmx
! ----------------------------------------
! AomBgmx = A-1B,Bu,gx (pour descente)
do k=1,nof
<;
! concatenate B mx and column-vectors
!-----------------------------------
  call omatcopy(no(k),mo(k)+moU,Bomx(kob(k)),no(k),
                                                AomBgmx(koamb(k)),no(k));
  call omatcopy(no(k),noxx,Gomx(io(k)+1,0),no(k),
                             AomBgmx(koamb(k)+no(k)*(mo(k)+moU)),no(k));
! [AmBGMx] = [AmB],[AmGamx]
!------------------------
  call sgesv(no(k),mo(k)+moU+noxx,Aomx(koa(k)),no(k),iPiv,
                                        AomBgmx(koamb(k)),no(k),infoa);
!  (copy dEta_dec back to Gamma for printout)
  ;call omatcopy(no(k),1,AomBgmx(koamb(k)+no(k)*(mo(k)+moU)),no(k),
                                                  GoMx(io(k)+1,0),no(k));
>;
 if noU.gt.0
 <
! concatenate B mx and column-vectors
!-----------------------------------
call omatcopy(noU,moU,Bomx(kob(u)),noU,AomBgmx(koamb(u)),noU);
call omatcopy(noU,noxx,Gomx(io(u)+1,0),noU,AomBgmx(koamb(u)+noU*moU),noU);
! [AmBGMx] = [AmB],[AmGamx]
!------------------------
call sgesv(noU,moU+noxx,Aomx(koa(u)),noU,iPiv,AomBgmx(koamb(u)),noU,infoa);
!  (copy dEta_dec back to Gamma for printout)
   call omatcopy(noU,1,AomBgmx(koamb(u)+noU*moU),noU, GoMx(io(u)+1,0),noU);
 >;
```

Les séquences donnent accès aux matrices du TEF, de descente et de couplage, les vecteurs Γ sont donnés en argument. On voit un parcours des familles pour ranger et calculer les

matrices descentes, puis le traitement éventuel d'une cellule sous U. On remarque *in fine* la copie dans **Gomx** de $\delta\eta_{dec} = A^{-1}\Gamma$ pour printout.

On a ensuite successivement trois calculs de $C^\dagger A^{-1} B$, nécessités par l'éclatement à faire des blocs $B_k, B_{k,u}$ d'une part, et le traitement particulier du transfert "externe". Bien évidemment, toutes les opérations portent sur **noxx** colonnes-vecteurs, c'est-à-dire que l'on effectue à la fois l'avance des états et transferts, et des sensibilités aux variables et paramètres.

En premier, $C^\dagger$ multiplie à la fois la matrice B double et le vecteur, pour les ranger dans $\mathbf{ComCpMx} = C_k^\dagger\{A_k^{-1}[B_k, B_{ku}, \Gamma_k]\}$, qui couplent chaque transfert interne à lui-même et à U.

```
! premier appel (IN,fm-Cells->IN,IX)
!-----------------------------------
! Coup_i,iu = I-D_i,u-CtAmB_i,u et Coup_i,x=Omega_i,x-CtAmG_i,x
do k=1,nof
<if (mo(k).gt.0)
 <
! [CtAmBGmx] = [Ct][AmBGmx]
! -------------------------
   call omatmlt(mo(k),no(k),mo(k)+moU+noxx,Comx(koc(k)),mo(k),
                           AomBgmx(koamb(k)),no(k),Comcpmx(kop(k)),mo(k));
;
!Rq: Coup_i,u = + [I - D_i] postponed as 4th
 >;
>;
```

On fait de même avec les blocs $C_{u,k}^\dagger$ du tableau **Comx(kocu(k))**, qui vont coupler entre-eux les composantes de $\delta\varphi_U$ via dernier bloc matriciel de **CoupUmx**, soit modifier les blocs $-D_{u,k}$ dans la liste des blocs démarrant **CoupUmx**.

```
! second appel (IX,Cellules->IN)
! -------------------------------
! CoupU_i = I-D_u,i-CtAmB_i    Premiers remplissages CoupU
! Rq si aucun IN-tr, pas de off-diag sur CoupU
do k=1,nof
<if mo(k).gt.0
 < call omatmlt(moU+mObs,no(k),mo(k),
                 Comx(kocu(k)),moU+mObs,AomBgmx(koamb(k)),no(k),
                                              CoupUmx(kodu(k)),moU+mObs);
  >;
>;
! troisieme appel (IX,Cellules->IX)
! ---------------------------------
! CoupU_i = [I-D_u]-Sum{'CtAmB_i,u'} et CoupU,x=[OmegU,x]-Sum{'CtAmG_i,x'}
! sum_up CtAmBu in diag part of CoupU
! -----------------------------------
call veczero(CoupUmx(kodu(u)),(moU+mObs)*(moU+noxx));
do k=1,nof
<if (mo(k).gt.0)
  call omatmltad(moU+mObs,no(k),moU+noxx,Comx(kocu(k)),moU+mObs,
                                 AomBgmx(koamb(k)+no(k)*mo(k)),no(k),
                                              CoupUmx(kodu(u)),moU+mObs);
>;
```

Après ces opérations, on a la structure matricielle suivante :

```
|  Am1B1    0                 | Am1B1u   contigues     dans AomBGmx(koamb(k))
                                          et suisvies de Gam,x
|  0     Am1B2                | Am1B2u
=============================================================================

| I-D1-Ct1Am1B1   0           |-D1u-Ct1Am1B1u   ensemble in ComCpmx(kop(i))

|  0    I-D2-Ct2Am1B2         |-D2u-Ct2Am1B2u
-----------------------------------------------------------------------------
|-[Du1+Cu1Am1B1, Du2+Cu2Am1B2 |I-Du-Sum_kCukAm1Bku     dans CoupUmx
```

(où on a reproduit les matrices descentes pour info). Il reste à traiter le cas de l'éventuelle présence d'une cellule sous U, dont l'elimination va directement coupler les compsantes du transfert externe :

```
! Cellule sous U
! --------------
if (noU.gt.0)
  call omatmltad(moU+mObs,noU,moU+noxx,Comx(kocu(u)),moU+mObs,
                                              AomBgmx(koamb(u)),no(u),
                                                  CoupUmx(kodu(u)),moU+mObs);
```

Alors, la partie de navette correspondant à "l'élimination des cellules" est terminée. Dans une arborescence à la ZOOM, il ne resterait que des familles et leurs transferts. Ici, il reste un seul niveau de familles (internes) et la famille externe sans cellule. Il restera à **EOLEF** à éliminer les tranferts internes pour obtenir l'équation de couplage sous l'Univers.

9.3 DownS

Cette routine effectue le calcul de $\delta\eta$ et extensions à partir de $\delta\varphi$. Voici l'appel et l'en-tête :

```
call DownS(nof,mObs,not,mot,noxx,noxx,odFimx,odEtamx,time,istep,Zprint,Zdebug);
;
subroutine DownS("in:" nof,mObs,not,mot,noxx,ndim,odFimx,
                 "out:" odetamx, "prnt:" time,istep,Zprint,Zdebug);
implicit none;
+SEQ,General_dim,Cl_mx,Tr_mx.
integer nof,ndim,u,k,noU;
real odFimx(mot+mObs,0:*),odEtamx(not,0:*);
! time values
real time; integer istep;
logical Zprint,Zdebug;
character*32 title;
;
! Initialisations
noU=no(nof+1); joU=jo(nof+1); moU=mo(nof+1); moI= mot-moU; u=nof+1;
```

Les calculs sont faits pour **ndim** colonnes de vecteurs — le vecteur d'avance d'état $\delta\eta$ et les sensibilités aux variables[1] $\delta\Phi$. On retrouve l'initialisation de paramètres de dimentionnement ayant vocation d'accroître la lisibilité.

[1] possiblement en plus les vecteurs de Floquet à "colonnes réelles".

Les cellules étant indépendantes et ayant chacune leur matrice de descente $A_k^{-1}B_k$ et $\delta\eta_dec = A_k^{-1}\Gamma_k$ rangés dans le tableau **AomBGmx** avec **kaomb(k)** comme pointeur, on traite arbitrairement la cellule sous **U** d'aborb quand elle existe, puis la cellule de chaque famille :

```
! U-Cell
! ------
if (noU.gt.0)
< call omatcopy(noU,ndim,AomBgmx(koamb(u)+noU*moU),noU,
                                                  odEtamx(io(u)+1,0),not);
  call omatmltsb(noU,moU,ndim,AomBgmx(koamb(u)),noU,
                                           odFimx(joU+1,0),mot+mObs,
                                                  odetamx(io(u)+1,0),not);
>;
! Fm Cells
!---------
! IX-tr first, then IN-tr
do k=1,nof
< call omatcopy(no(k),ndim,AomBgmx(koamb(k)+no(k)*(mo(k)+moU)),no(k),
                                                  odEtamx(io(k)+1,0),not);

  call omatmltsb(no(k),moU,ndim,AomBgmx(koamb(k)+no(k)*mo(k)),no(k),
                                             odFimx(joU+1,0),mot+mObs,
                                                  odEtamx(io(k)+1,0),not);
  if (mo(k).gt.0)
   call omatmltsb(no(k),mo(k),ndim,AomBgmx(koamb(k)),no(k),
                                           odFimx(jo(k)+1,0),mot+mObs,
                                                  odEtamx(io(k)+1,0),not);
>;
return;
end;
```

Dans chaque famille k, on a

$$\delta\eta_k = \delta\eta_k^{dec} - [A^{-1}B]_k\delta\varphi_U - [A^{-1}B]_k \mid \delta\varphi_{k,u} > -[A^{-1}B]_k \mid \delta\varphi_k >$$

c'est-à-dire que seul le transfert externe est connecté à toutes les cellules, alors que chaque cellule ne voit que le transfert interne éventuel à sa famille.

9.4 L'avance d'un pas de temps remplaçant OKER

Pour l'instant au moins, on ne rassemble pas ce code dans une routine du genre **oker**. Une navette se compose d'une montée (**MOVUP**), d'une résolution sous **U**, puis d'une descente (**MOVDWN**) :

1. élimination des $\delta\eta_k$ avec conservation des matrices et vecteurs de redescente (**EOLES**) ;
2. élimination des $\delta\varphi_k$ avec conservation des matrices et vecteurs de redescente (**EOLEF**) ;
3. résolution de l'équation matricielle $\delta\varphi_U = U\delta\varphi_U^{ins}$ (**EOLEMAX**) ;
4. calcul des $\delta\varphi_k$ des familles internes (**DOWNF**) ;
5. calcul des $\delta\eta_k$ (**DOWNS**) ;

La montée d'abord :

```
! ++++++++++++++++++++++
!    MOVUP
! ++++++++++++++++++++++
!        EOLES
! ++++++++++++++++++++++
call EoleS(nof,mObs,not,mot,noxx,noxx,GoMx,time,istep,iPiv,Zprint,Zdebug);
! ++++++++++++++++++++++
!        EOLEF
! ++++++++++++++++++++++
call EoleF(nof,mObs,mot,noxx,noxx,oMgmx,time,istep,iPiv,Zprint,Zdebug);
```

alors, on a une matrice dîte de couplage (**CoupU**) et un vecteur connu sous U.

On recopie ce vecteur connu des tableaux de montée dans le tableau **odFi**, avant de résoudre :

```
! ++++++++++++++++++++++
!    EOLEMAX
! ++++++++++++++++++++++
! copy first deta_ins and dfi_ins in dEta dFi
! ------------------------------------------
do k=1,nof
< if (mo(k).gt.0) call omatcopy(mo(k),noxx,
       ComCpmx(kop(k)+mo(k)*(mo(k)+moU)),mo(k),odFimx(jo(k)+1,0),mot+mObs);
>;
! idem sous U
! -----------
 call omatcopy(moU+mObs,noxx,CoupU(1,mot+1),moU+mObs,
                                                  odfimx(joU+1,0),mot+mObs);
! solve U system
! --------------
 call sgesv(moU+mObs,noxx,CoupUmx(kodu(u)),moU+mObs,iPiv,
                                          odfimx(joU+1,0),mot+mObs,infores);
 if (infores.ne.0) print*,'*** SGESV error in U-system: ',infores;
```

on a alors la solution $\delta\varphi_U$ (et ses extensions).

Il faut à présent calculer les $\delta\varphi_k$ de chaque famille interne :

```
! ++++++++++++++++++++++
!    MOVDWN
! ++++++++++++++++++++++
!        DOWNF
! ++++++++++++++++++++++
do k=1,nof
< if (mo(k).gt.0)
   call omatmltsb(mo(k),moU,noxx,ComCpmx(kop(k)+mo(k)*mo(k)),mo(k),
                                           odfimx(joU+1,0),mot+mObs,
                                            odfimx(jo(k)+1,0),mot+mObs);
>;
! ++++++++++++++++++++++
!        DOWNS
! ++++++++++++++++++++++
call DownS(nof,mObs,not,mot,noxx,noxx,odFimx,odEtamx,time,istep,Zprint,Zdebug);
! ==========================
! End of step advance for dEta
```

```
! ============================
! (Tr material will be updated after D-loop at beg of nxt step in time-loop)
```

puis il reste à **DownS** le calcul des avances d'état, comme ci-dessus.

Il reste à sortir ces informations en listing du run, et on termine par le calcul des nouveaux vecteurs η_k et φ_0, ce dernier n'étant qu'un estimation linéaire de $\varphi = f(\eta, \varphi)$, servant à initialiser la **D_loop** :

```
if Zprint
< print*;
  print*,'>>>>>>>>  time advance:',time,' dt= ',dt,' step:',istep;
  print*,'> Family   Var_Name          Eta          dEta_dec    dEta';
  do k=1,nof+1
  < if no(k).gt.0
    < i=io(k)+1;
      write(*,9002) FmNam(k)(1:12),EtaNam(i)(1:16),Eta(i),Gamma(i),odEta(i);
      do i=io(k)+2,io(k)+no(k)
      < write(*,9001) EtaNam(i)(1:16),Eta(i),Gamma(i),odEta(i);
      >;
    >;
  >;
>;
! Advance in Eta, guess in Phiz
! -----------------------------
do i=1,not*noxx< Eta(i)=eta(i)+odEta(i); >;
do j=1,mot< phiz(j)=ff(j)+odfi(j); >;
```

Comme dans la **102**, des calculs supplémentaires sont effectués pour déterminer les "probes" et la sensibilité aux paramètres. Ces calculs, de même que la détermination des sources de sensibilité sont décrits ailleurs.

9.5 La D_Loop

On la donne ici, la navette associée étant la précédente amputée des équations d'état, puisque η étant donné, on doit résoudre l'équation implicite $\varphi = f(\eta, \varphi)$. L'unique différence avec la **102** est liée à l'emboîtement.

La solution est cherchée en **ff(.)**, que l'on initialise avec φ_0 estimé par son avance linéaire par $\varphi + \delta\varphi$. Ceci étant fait, on calcule ff avec les formules de l'utilisateur, mais pour synchroniser les passages de φ_0 à φ, on calcule $\varphi = f(\eta, \varphi_0)$, ce petit tour de passe-passe étant l'objet des macros de substitution encadrant ces calculs. La super-macro **mult** construit une liste **ff(1)=..., ff(2)=...** comme on l'a déjà expliqué dans le passage sur les macros **Set_Eta_Phi**. On compte les itérations dans **ntrial** :

```
! ****************************
!          D-Loop
! ****************************
ntrial=0;errx=obig;
:D_iter:; [ntrial:+1]; errxbef=errx;
!--------------------------
! Calcul de ff = f(eta,Phiz)
!--------------------------
&'ff(#)'='phiz(#1)' &'phiz(#)='='ff(#1)=#N'
&'(#ff(#)#)'='(#1phiz(#2)#3)' &'(phiz(#))(/ff('='(ff(#1))(/ff('
;mult [j]=1,momlt : ff[j] = Phi_tef[j];
```

```
!---------------------------------
! Matrices d(i,j) = dFi(i)/dPhiz(j)
!--------------------------------- (inclue mObs)
          ;mult [i]=1,momlt : d([.i],j) = f_d(Phi_tef[i])(/ff(j));
          ;mult [i]=1,momlt : d([.i],j) = f_d(Phi_tef[i])(/ff(jou+j));
&'#Kff(#)' &'#K(#ff(#)#)' &'#Kphiz(#)=' &'#K(phiz(#))(/ff(';
```

Il a fallu bien sûr recalculer les matrices D pour prendre en compte les nouveaux états et les solutions provisoires en φ_0.

Il reste à monter une courte navette qui élimine les tranferts internes du système $(I - D)\delta\varphi = f(\eta, \varphi_0) - \varphi_0$:

```
! raz Comcpmx and CoupUmx
! ------------------------
 call veczero(Comcpmx,noptot); call veczero(CoupUmx,(moU+mObs)*(mot+1));
;
! residue ff-phiz in Omega
! ------------------------
 call omatcopy(mot,1,ff(1),mot+mObs,                        Omgmx(1,0),mot+mObs);
 call omatsub (mot,1,Omgmx(1,0),mot+mObs,phiz(1),mot,    Omgmx(1,0),mot+mObs);
!-------------------
! EoleMax in D_loop
!-------------------
! copy first dfi_ins=ff-phiz in dFi
! call k2copy(nof,mo,1,ComCpmx,kop,?offset_Tr,odFi,jo,+1,mo); ???
do k=1,nof
<
  if (mo(k).gt.0) call omatcopy(mo(k),1,
            ComCpmx(kop(k)+mo(k)*(mo(k)+moU)),mo(k),odFi(jo(k)+1),mo(k));
>;
 call omatcopy(moU+mObs,1,CoupU(1,mot+1),moU+mObs,
                                                  odFi(joU+1),moU+mObs);
! solve U system
! --------------
 call sgesv(moU,1,CoupUmx(kodu(u)),moU+mObs,iPiv,
                                           odFi(joU+1),moU+mObs,infores);
  if infores.ne.0< print*,'*** SGESV error in  D_loop: ',infores; >;
!-----------------
! MovDwn
!-----------------
do k=1,nof
< if (mo(k).gt.0)
   call omatmltsb(mo(k),moU,1,ComCpmx(kop(k)+mo(k)*mo(k)),mo(k),
                                                 odfi(joU+1),moU+mObs,
                                                      odfi(jo(k)+1),mo(k));
>;
```

élimination ayant permis de résoudre le système couplé sous **U** et redescendre dans les familles.

La fin d'une itération consiste à évaluer le résidu $||\varphi - \varphi_0||$:

```
errx=0.;<i=1,mot; [errx:+abs(odFi(i))]; >;
<i=1,mot; phiz(i)=phiz(i)+odFi(i); >;
if (errx.gt.tolx.and.ntrial.le.10.and.errx.lt.errxbef) goto :D_iter:;
```

```
if Zprint
< print*,'####> D_Loop ended after',ntrial,' iterations; residual error:',errx;
  print*,'_____________________________________________________________';
>;
if (errx.gt.tolx)
print*,'***** WARNING no convergence in D_loop, residual=',errx,
'after: ',ntrial,'iterations';
! ============== end of D_Loop ====================
```

On boucle 10 fois (ici) si non-convergence :**goto :D_iter :**.

Il est possible de rassembler ces calculs de navette dans une routine (**Solve_D ?**).

9.6 Calculs liés aux sensibilités faisant usage de EOLEF

La gestion différenciée des sources et réaffectation de Φ pose plusieurs calculs faisant intervenir $(I-D)^{-1}$. Posons pour faire complet les calculs mettant à jour ces sources et ré-initialisations :

1. **pert type**, initialisation de $\Phi_k(t) = 0$;
2. **pert type** (pulse), initialisation de $\Phi_k(t) = Bb(I-D)^{-1} \mid u_k >$;
3. **pert type** (step $\frac{1}{1-g}$), ajout de sources unité au calcul de $(I-D)\Psi = C^\dagger \Phi + \mid u_k >$;
4. **pert type** (step $\frac{g}{1-g}$), ajout de sources au calcul de

$$\Gamma_\Phi = dt \ \left[\ H\Phi + B_b\Psi + B_b(I-D)^{-1} \mid u_k > \right]$$

(où on utilise encore les vecteurs de la base euclidienne $\mid u_k >$, avec $U = I$). La source de type 3 est simplement absorbée dans le calcul de Ψ. Il ne sert à rien d'essayer de judicieusement regrouper les calculs des types 2 et 4 en résolvant $(I-D)\delta\Psi = \mid u_k >$ pour l'ensemble des occurences des sources unité des types 2 et 4 (une seule navette). Ceci parce que le type 2 n'est fait qu'au passage par un ionstant de ré-initialisation dnas Psteer, alors que le type 4 mène à une source perpétuelle, à inclure dans le Γ_Φ.

Pour le type 4 que l'on décrit ici, on résoud $(I-D)\delta\Psi = \mid u_i >$ auquel on ajoute $\Psi + \delta\Psi \to \delta\Psi$, avant de calculer

$$\Gamma_\Phi = dt \ \left[\ H\Phi + B_b\delta\Psi \ \right]$$

. On effectue alors une seule navette des produits par B_b :

```
! Gam Phi = dt[ H Phi + Bb Psi ] [+source pert 4]
! ----------------------------------------------
  if nox.gt.0
  <;call veczero(odPsimx(1,1),(mot_mObs)*nox); Zflag=.false.; "calcul local"
! Perpetual Source on Phi_t = Bb(I-D)m1 {|u_i>} first
! --------------------------------------------------
! -- selection des |u_i>
    Do ko=1,nox
    < if istep.ge.ko_p_ini(ko).and.ko_p_typ(ko).eq.4
       < Zflag=.true.; odPsimx(ko,ko_p_kvar(ko))=1.D0;  "=|U_i>" >;
    >;
! -- resolution de (I-D)X={|u_i>}
    if Zflag
    < call EoleF(nof,mObs,mot,noxx,nox,odPsimx(1,1)
                                              ,time,istep,iPiv,Zprint,Zdebug);
    >;"end Zflag"
```

```
! -- add Psi to dPsi (zero default) dPsi = Psi [+ (I-D)m1|u_i> ]
   call omatadd(mot,nox,oPsimx(1,1),mot+mObs,odPsimx(1,1),mot+mObs
                                                    ,odPsimx(1,1),mot+mObs);
```

On a alors en sortie soit $\delta\Psi$ nul, soit avec la bonne source, et on y a ajouté Ψ, ce qui permet un seul calcul de produit par B_b :

```
! -- Gam Phi = dt[ H Phi + Bb dPsi ]
    do k=1,nof
    < call omatmlt(no(k),no(k),nox,Homx(koa(k)),no(k),oPhimx(io(k)+1,1),not
                                                    ,GoMx(io(k)+1,1),not);
      if (mo(k).gt.0)
       call omatmltad(no(k),mo(k),nox,Bobmx(kob(k)),no(k)
                          ,odPsimx(jo(k)+1,1),mot+mObs ,GoMx(io(k)+1,1),not);
      call omatmltad(no(k),moU,nox,Bobmx(kob(k)+no(k)*mo(k)),no(k)
                            ,odPsimx(joU+1,1),mot+mObs ,GoMx(io(k)+1,1),not);
    >;
    if noU.gt.0
    < call omatmlt(noU,noU,nox,Homx(koa(U)),noU,oPhimx(io(U)+1,1),not
                                                    ,GoMx(io(U)+1,1),not);
      call omatmltad(noU,moU,nox,Bobmx(kob(U)),noU
                            ,odPsimx(joU+1,1),mot+mObs ,GoMx(io(U)+1,1),not);
    >;
  >;"End nox"
```

avec le traitement final d'une éventuelle cellule sous U — qui ne voit que des transferts "externes".

9.7 Calcul de la matrice d'avance de phase : une navette à part

La matrice d'avance de phase — dîte **aspha** — est calculée, malgré son coût pouvant être élevé, pour conserver les mêmes facilités que la 101. Elle se calcule par élimination des transferts : $A^\sharp = H + B_b(I-D)^{-1}C^\dagger$. On a deux manières d'effectuer ce calcul, soit on élimine d'abord les transferts internes, soit les externes. Il faudrait choisir en fonction des dimensions, mais on n'a codé pour l'instant que la méthode par élimination des internes d'abord, en supposant que leur dimension cumulée sera en général plus grande que celle des externes — ce qui ne sera pas le cas du modèle bi-dim[1].

On regroupe ces calculs faisant usages de matrices supplémentaires au TEF dans la routine **doAspha**. Ces tableaux étant :

1. **oDscMx** descente, semblables à **AomBgmx**, qui permet le calcul de $(I - D_i)^{-1}[D_{iu}, Ct_i]$;
2. les trois tableaux **owBu, owCu, oWDu** qui recopient d'abord les trois matrices jacobiennes liées à l'Univers, et sont ensuite modifiées par l'ajout de termes d'élimination des internes ;
3. **aspha** joue alors le rôle de **CoupUmx** ;

[1] ce choix a de plus l'énorme avantage de s'appliquer *mutatis mutandis* à la navette du calcul de la matrice réduite, cf section suivante.

On peut imaginer les opérations de complémentation de Schur sur la structure matricelle suivante, qui a réordonner les objets pour procéder à l'élimination en descendant la diagonale principale :

```
 | D1          | D1u | Ct1          => oDscmx
 |             |     |
 |     D2      | D2u |     Ct2
 |             |     |
 |         D3  | D3u |         Ct3
 |---------------------------------
 |-Du1-Du2-Du3 | I-Du| Cu1 Cu2 Cu3   => owDu et ow Cu
 |---------------------------------
 |  B1         | B1u | H1
 |             |     |
 |     B2      | B2u |      H2       => aspha
 |             |     |
 |         B3  | B3u |           H3
 |                   \- => owBu
```

L'appel et l'en-tête de doaspha d'abord, avec un calcul de dimension totale des matrices descente, qui seront volatiles dans doaspha :

```
! ********************************************************************
!  Calcul de la matrice d'avance dans l'espace de phase  [ASPHA]
!       d_t Eta = [ A + B.(I-D)-1.Ct ] Eta
! ********************************************************************
ko=0; <k=2,nof+1; [ko: + mo(k-1)*(mo(k-1)+moU+no(k-1))];>;
call DoAspha("in:" nof,mObs,not,mot,moU,ko,
                "out:" aspha, "prt:" time,istep,iPiv,Zprint,Zdebug);
! --------------------------------------------------------------------
;
subroutine DoAspha("in:" nof,mObs,not,mot,moU,ndsc,
                   "out:" aspha, "prt:" time,istep,iPiv,Zprint,Zdebug);
implicit none;
+SEQ,General_dim,Cl_mx,Tr_mx.
integer nof,ndsc,u,k,noU;
real Aspha(not,not);
real owDu(moU,moU),owBu(not,moU),owCu(moU,not),owDscmx(ndsc);
integer kowd(nof+1);
! time values
real time; integer istep;
logical Zprint,Zdebug;
character*32 title;
! LaPack
integer iPiv(mot),infoa;
;
if Zdebug<print*,'==========';print*,' Aspha';print*,'==========';>;
! Initialisations
noU=no(nof+1); joU=jo(nof+1); moI= mot-moU; u=nof+1;
;
! provisoire :
call veczero(aspha,not*not);
```

```
kowd(1)=1;
do k=2,nof+1
< kowd(k)=kowd(k-1)+mo(k-1)*(mo(k-1)+moU+no(k-1));
>;
%
```

Les autres matrices de travail ont des dimensions fixées par les paramètres standards. On calcule aussi un pointeur sur ces matrices descente de chaque famille. On iniitialise ces matrices descentes avec les $I-D, -D$ blocs matrices, avant de calculer $-(I-D_i)^{-1}D_{i,u}$:

```
! matrices descentes de IN a IX
! ------------------------------
! <a> forme [I-Di],Diu,-Ctiu -> [I-Di]m1*...
do k=1,nof
< if mo(k).gt.0
  <
    call omatcopy(mo(k),mo(k)+moU,Domx(kod(k)),mo(k),
                                                  owDscmx(kowd(k)),mo(k));
    call omatsb2U(mo(k),mo(k)+moU,owDscmx(kowd(k)),mo(k),
                                                  owDscmx(kowd(k)),mo(k));
    call omatcopyN(mo(k),no(k),Comx(koc(k)),mo(k),
                                owDscmx(kowd(k)+mo(k)*(mo(k)+moU)),mo(k));
    call sgesv("in:" mo(k),moU+no(k),owDscmx(kowd(k)),mo(k),iPiv,
                     "in/out" owDscmx(kowd(k)+mo(k)*mo(k)),mo(k),infoa);
    if Zdebug
    < TITLE='owDscmx: final';CALL printmat2(title,mo(k),mo(k)+moU+no(k),
                                owDscmx(kowd(k)),mo(k),0,2,k);
    >;
  >;
>;
```

On peut alors démarrer l'élimination, on a choisit en premier d'abonder **owDu** avec $I-D_u+\sum_k D_{u,k}(I-D_k)^{-1}D_{k,u}$, avec les bons signes ...

```
! <b> IX,fm,IX  -> Du
! -----------------
call omatcopy(moU,moU,Domx(kod(u)),moU+mObs,owDu,moU);
call omatsb2U(moU,moU,owDu,moU,owDu,moU);
do k=1,nof
< if (mo(k).gt.0)
  call omatmltad(moU,mo(k),moU,Domx(kod(U)+kodu(k)-1),moU+mObs,
                               owDscmx(kowd(k)+mo(k)*mo(k)),mo(k),owDu,moU);
>;
! <c> IX,IN,IN  -> Cu
! -----------------
call omatcopyN(moU,not,Comx(koc(U)),moU+mObs,owCu,moU);
do k=1,nof
< if (mo(k).gt.0)
  <  call omatmltad(moU,mo(k),no(k),Domx(kod(U)+kodu(k)-1),moU+mObs,
             owDscmx(kowd(k)+mo(k)*(mo(k)+moU)),mo(k),owCu(1,io(k)+1),moU);
  >;
>;
```

et en second on a remplit la bande **owCu** avec $-Ct_u+\sum_k D_{u,k}(I-D_k)^{-1}C^{\dagger}_{k,u}$.

On passe ensuite aux produits $B_b(I-D)^{-1}D$ de la bande **owBu** :

```
! <d> Cl,IN,ClX -> Bu
! -----------------
do k=1,nof
< call omatcopy(no(k),moU,Bobmx(kob(k)+no(k)*mo(k)),no(k),
                                                    owBu(io(k)+1,1),not);
>;
if (noU.gt.0)
 call  omatcopy(noU,moU,Bobmx(kob(U)),noU,          owBu(io(U)+1,1),not);
;
do k=1,nof
< if (mo(k).gt.0)
   call omatmltsb(no(k),mo(k),moU,Bobmx(kob(k)),no(k),
                  owDscmx(kowd(k)+mo(k)*mo(k)),mo(k),owBu(io(k)+1,1),not);
>;
if Zdebug<TITLE='owBu: final';CALL printmat2(title,not,moU,owBu,not,2,1,1)>;
;
! <e> Cl,IN,IN  -> Aspha
! --------------------
do k=1,nof
< call omatcopy(no(k),no(k),Homx(koa(k)),no(k),Aspha(io(k)+1,io(k)+1),not);
  if (mo(k).gt.0)
   call omatmltsb(no(k),mo(k),no(k),Bobmx(kob(k)),no(k),
      owDscmx(kowd(k)+mo(k)*(mo(k)+moU)),mo(k),aspha(io(k)+1,io(k)+1),not)>;
if (noU.gt.0)
 call omatcopy(noU,noU,Homx(koa(U)),noU,Aspha(io(U)+1,io(U)+1),not);
if Zdebug
 <TITLE='Aspha: avant IX,IX';CALL printmat2(title,not,not,Aspha,not,1,1,1)>;
;
! <f> [I-Du...]m1*Cu
 call sgesv("in:" moU,not,owDu,moU,iPiv,           "in/out" owCu,moU,infoa);
```

alors que les produits $B_b(I-D)^{-1}C^\dagger$ vont directementg dans l'aspha.

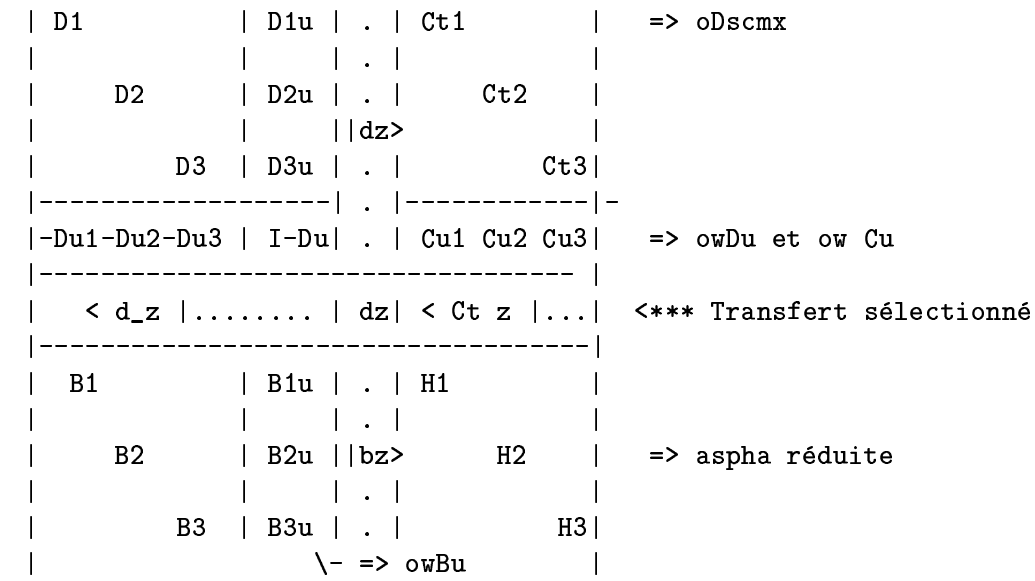
Il ne rest plus qu'à éliminer les transferts externes, ce qui se fait en une seule oipération :

```
! <g> IX,u,IX -> aspha
! --------------------
call omatmltsb(not,moU,not,owBu,not,owCu,moU,aspha,not);
;
if Zdebug<TITLE='Aspha: final';CALL printmat2(title,not,not,Aspha,not,1,1,1)>;
;return;
end;
```

ce qui termine le calcul de l'$A^\sharp$ — qui ne dépend pas, rappelons-le, du pas de temps, malgré cette appellation adoptée par l'unsecte.

9.8 Calcul dans la Boreleig : une autre navette à part

Dans la Boreleig, on doit en plus de la A^{spha} calculer une matrice d'avance réduite $A^\flat$ pour le gain de rétroaction. On doit alors définir un système équivalent avec un seul transfert (scalaire) pointé dans ZINIT, ce qui conduit à réarranger le système comme suit :



Il s'agit ainsi de conduire une élimination de tous les transferts — le système correspondant étant amputé d'unbe ligne et d'une colonne. L'élimination est alors semblable au calcul de la A^{spha} avec arrêt au dernier transfert. L'agorithme se présente ainsi :

1. identification de la famille du transfert pointé par **koindex** ;
2. suppression simultanée d'une colonne de D et copies dans $\mid d_z >$, $< d_z \mid$ et d_z ;
3. suppression d'une colonne de B et copie dans $\mid b_z >$;
4. suppression d'une ligne de C^t et copie dans $< c_z \mid$;
5. élimination par appel à **doaspha** particularisé du fait d'une réduction des dimensions des Jacobiennes liées au transfert sous **Zeus** ;

Il s'agit comme pour la Boreleig de la 102, d'un jeu pénible d'indices, aggravé par la partition. C'est ce jeu qu'il faut à présent jouer en finesse.

1. détection de la famille : joindex compris entre io(k) et io(k+1) donne $k_{index} := k$;
2. augmentation de moU $\rightarrow mo U+1$, c'est-à-dire que l'on augmente les dimensions des matrices U ;
   - $k = u$
     - balayage dans les familles :
     - matrice $D_{k,u}$: copie dans $\mid d_z >$ et suppression de la colonne ;
     - matrice $B_{k,u}$, même opération ;
     - sous U, on déplace $< d_z \mid$ et $< C_z^t \mid$, et suppressions ;

     L'élimination peut se scinder en deux : dans un premier temps, on forme les B_{bi} et $D_{iu}(I - D_i)^{-1}$ fois D_{iu} ou $C_i^\dagger$ sans changer d'ordre, puis on effectue le réarrangement d'ordre dans $D_u, C_u^\dagger$ et B_u. On ne devrait donc pas, dans ce cas, à augmenter la dimension des matrices U, mais comme il faut le faire à la compilation, on utilisera ces lignes-colonnes pour les matrices du transfert, et on remplacera les anciennes par des lignes/colonnes bidons (zero et un à leur inetrsection) de manière à standardiser la denière élimination sous **U**..
   - $k \neq u$

– dans les familles k_{index} :
– matrice D_k : copie dans $\mid d_z >$ et $< d_z \mid, d_z$ et suppressions ;
– matrice B_k, même opération pour $\mid b_z >$ en dernière colonne ;
– sous U, on déplace $\mid d_z >$ et suppression d'une colonne ;
Mais dans ce cas, il faut procéder au changement d'ordre avant toute élimination. On va mettre les lignes et colonnes **joindex** en ligne-colonne supplémentaires aux matrices U, et en "coiffer" $A^\flat$ par un jeu d'équivalence. Les matrices à gauche des produits de complémentation $B_{joindex}$ et $D_{u,joindex}$ seront stockée réarrangées dans **Bbuf** et **Dbuf**.

Par rapport à **DoAspha** donc, on a d'abord tous les rangements, avec réarrangements pour $k \neq u$, et l'élimination se produit avec les cas deux cas particuliers portant sur les dimensions et un réordonnancement avant deuxième élimination pour $k = U$.

On crée une nouvelle routine **matcopX** qui recopie un bloc matriciel á l'exception d'une ligne et d'une colonne dans un bloc où elles seront consécutives. On a aussi la routine qui échange deux colonnes ou deux lignes d'une matrice (col & raw swap).

Pour détailler les instructions, on va distinguer les deux cas du tranfert dans U ou non[1].

déclarations des matrices et initialisations. On effectue ce cadrage des matrices avec **bspha** de manière que les $d_z, \mid b_z >$ et $< c_z \mid$ soient sur les bords gauche et haut de **bspha(0,0)**. De même, on cadre les matrices sous **U** autour de bspha. Le jeu d'équivalence devrait faire que le tableau des matrices ci-dessus corresponde à un tableau Fortran pour sa partie sous U.

```
real owDu(moU+1,moU+1),owBu(not,moU+1),owCu(moU+1,not),owDscmx(ndsc);
integer kowd(nof+1); "ptr in owDscmx"
real bspha(0:not,0:not),d_z,b_z(not),c_z(not);
! cadrage des matrices pour que bspha soit bordee en haut et a gauche
! par d_z, <c_z| et |b_z>
equivalence (owDu(moU+2,1),owBu(1,1),(owDu(1,moU+2),owCu(1,1)),
            (owDu(moU+1,moU+1),bspha(0,0);)
equivalence (d_z,owDu(moU+1,moU+1),(b_z,bsapha(1,0),(c_z,bspha(0,1);
real Dubuf(moU=1),mot-moU); "Du mx band left of owDu"
real Bbuf(not,mot-moU);      "one Bb-mx left of owBu"
; . . .
! Initialisations
noU=no(nof+1); joU=jo(nof+1); moI= mot-moU; u=nof+1; moZ=moU+1;
!bemol detection famille joindex
<k=1,nof+1; if joindex.le.(jo(k)+mo(k)) goto :fm:; >;
print*,'*** error joindex impossible:',joindex,joU+moU; stop 'Boreleig';
:fm: kofm=k; if (kofm.eq.u) ZphiU=.true.;
```

Mais ces équivalences sont interdites en Fortran sur des tableaux locaux volatiles ; on effectue alors cette équivalence par les substitutions de variables suivantes, qui auront pour effet de travailler en réalité dans le tableau **bspha**, à présent explicitement encadré avec des bandes ligne et colonne d'indices négatifs :

```
&F;
       real bspha(-moU:not,-moU:not)
&M;
!== real owDu(moU+1,moU+1),owBu(not,moU+1),owCu(moU+1,not);
```

[1] ce qui suggère peut-être une autre écriture effectuant proprement cette séparation.

```
!== real d_z,b_z(not),c_z(not);
! cadrage des matrices pour que bspha soit bordee en haut et a gauche
! par d_z, <c_z| et |b_z>
!== equivalence (owDu(moU+2,1),owBu(1,1)),(owDu(1,moU+2),owCu(1,1)),
!==             (owDu(moU+1,moU+1),bspha(0,0));
!== equivalence (d_z,owDu(moU+1,moU+1)),(b_z,bspha(1,0)),(c_z,bspha(0,1));
&'owDu'='bspha(-moU,-moU)' &'owDu(#,#)'='bspha(-moU-1+#1,-moU-1+#2)'
&'owBu'='bspha(1,-moU)' &'owBu(#,#)'='bspha(#1,-moU-1+#2)';
&'owCu'='bspha(-moU,1)' &'owCu(#,#)'='bspha(-moU-1+#1,#2)'
&'d_z'='bspha(0,0)' &'b_z'='bspha(0,1)' &'b_z(#)'='bspha(0,#1)'
&'c_z'='bspha(1,0)' &'c_z(#)'='bspha(#1,0)'
```

On remarque la déclaration en Fortran de **bspha**, obligée à cause de l'effet de deux signes " :" dans la même instuction, marquant un label pour Mortran — ce type de déclaration n'existait pas à la création de Mortran.

Il faut encore préparer deux buffers pour le cas transfer interne. En effet, ssa matrice B_b devra être amputée d'une colonne i — on dimensionne large, mais on n'a guère le choix[1]. Pour ses blocs D, D_{iu}, la colonne a déplacer et la matrice résiduelle seront copiées dans les **owDscmx** , mais on devra construire une bande **Dbuf** pour recopier une ligne en dernière en supplément des D_{ui}.

Le pointeur des blocs dans **owDscmx** tiend compte de la réduction de dimension dù à l'enlèvement des ligne/colonne.

On peu à présent détailler les calculs.

transfert sous U. Dans ce cas, les copies sont comme dans **DoAspha**, de même que la première série d'éliminations[2], que nous ne détaillerons qu'ensuite :

```
if ZPhiU
< do k=1,nof
  < "<a>"
    if mo(k).gt.0
    < call omatcopy(mo(k),mo(k)+moU,Domx(kod(k)),mo(k),
                                                    owDscmx(kowd(k)),mo(k));
      call omatcopyN(mo(k),no(k),Comx(koc(k)),mo(k),
                                  owDscmx(kowd(k)+mo(k)*(mo(k)+moU)),mo(k));
    >;
  >;
! <b>
  call omatcopy(moU,moU,Domx(kod(u)+moI*(moU+mObs)),moU+mObs,owDu,moZ);
! <c>
 call omatcopyN(moU,not,Comx(koc(U)),moU+mObs,owCu,moZ);
! <d> et <e>
do k=1,nof
< call omatcopy(no(k),moU,Bobmx(kob(k)+no(k)*mo(k)),no(k),
                                                          owBu(io(k)+1,1),not);
  call omatcopy(no(k),no(k),Homx(koa(k)),no(k),bspha(io(k)+1,io(k)+1),not);
>;
if (noU.gt.0) call omatcopy(noU,moU,Bobmx(kob(U)),noU,  owBu(io(U)+1,1),not);
```

La particularité ici est d'effectuer une permutation avant la dernière élimination :

[1] c'est mauvais, il faudrait mieux faire avec une subroutine genre omatmltadX, qui excluera une ligne et/ou une colonne, le problème étant qu'on ne veut pas détruire les Jacobiennes calculées dans **principal**.

[2] Attention, les dimensions des matrices sous U sont restées en oubliant qu'elles seront remplacées par bspha, dont le nombre de colonnes est **not+moZ** en réalité.

```
!bemol: permutation pour ZPhiu avant derniere elimination
if ZPhiU
< call colswap(moZ+not,joindex-moI,moZ,owDu,moZ+not);
  call rawswap(moZ+not,joindex-moI,moZ,owDu,moZ+not);
>
! Elimination partielle sous U pour laisser d_z, <c_z| et |b_z>
! <f> [I-Du...]m1*Cu
call sgesv("in:" moU,not,owDu,moU,iPiv,           "in/out" owCu,moU,infoa);
! <g> IX,u,IX -> bspha
! ------------------
call omatmltsb(not,moU,not,owBu,not,owCu,moU,bspha,not+1);
```

Après l'échange des lignes et colonnes par les **swap** routines, on a une ligne et une colonne pleine de zéro à l'exception de leur intersection qui vaut 1. Ainsi, la dernière élimination porte sur les **moU** colonnes de owBu, et modifie à la fois **aspha := bspha(1,1)** et la première colonne **bspha(.,0)**, de même pour sa premiére ligne, dans lesquelles, ainsi, on trouvera les pseudo-jacobiennes du système uni-transfert : $d_z, \mid b_z >$ et $< c_z \mid$.

transfert sous une famille. On a d'abord la construction réduite pour l'une des **owDscmx**, les matrices descente :

```
< do k=1,nof
  < "<a>"
    if mo(k).gt.0
    < if k.ne.kofm                                                          *
      < call omatcopy(mo(k),mo(k)+moU,Domx(kod(k)),mo(k),                   *
                                                owDscmx(kowd(k)),mo(k));    *
        call omatcopyN(mo(k),no(k),Comx(koc(k)),mo(k),                      *
                                  owDscmx(kowd(k)+mo(k)*(mo(k)+moU)),mo(k));*
      >else "joindex in fm"
      < kloc = joindex-jo(k);
        call omatcopX(mo(k),mo(k)+moU,kloc,kloc,
                               Domx(kod(k)),mo(k),owDscmx(kowd(k)),mo(k)-1);
! (copie a l'exclusion de ligne arg1, col arg2)
        call omatcopXN(mo(k),no(k),kloc,0,Comx(koc(k)),mo(k),
                             owDscmx(kowd(k)+(mo(k)-1)*(mo(k)+moU)),mo(k)-1);
! copies dans les U-mx
! --------------------
!  les Dk,ku Domx
        d_z = Domx(kod(k)+mo(k)*(kloc-1)+kloc-1);
! copie deux semi-colonnes haut et bas dans owDscmx et Du, col de droite
        call omatcopX(mo(k),1,kloc,0,Domx(kod(k)+(kloc-1)*mo(k)),mo(k),
                           owDscmx(kowd(k)+(mo(k)-1+moU)*(mo(k)-1)),mo(k)-1);
        call omatcopy(moU,1,Domx(kod(u)+kodu(k)-1+(kloc-1)*(moU+mObs)),
                                         moU+mObs,     owDu(1,moZ),not+moZ);
! copie deux semi-lignes  dans dubuf et owDu, lignes du bas
        call omatcopXN(1,mo(k),0,kloc,Domx(kod(k)+kloc-1),mo(k),
                                                      Dubuf(moZ,1),moZ);
        call omatcopy(1,moU,Domx(kod(k)+mo(k)*mo(k)+kloc-1),mo(k),
                                                    owDu(moZ,1),not+moZ);
! et complement Du,k -> Dubuf
        call omatcopXN(moU,mo(k),0,kloc,Domx(kod(u)+kodu(k)-1),moU+mObs,
                                                           Dubuf,moZ);
```

```
! la Bobmx
! and now Bbuf = Bb sauf |b-z> -> left of owBu and Bbuf
        call omatcopX(no(k),mo(k),0,kloc,Bobmx(kob(k)),no(k),   Bbuf,no(k));
        call omatcopy(no(k),1,Bobmx(kob(k)+(kloc-1)*no(k)),no(k),
                                                  owBu(io(k)+1,moZ),not+moZ);
! une ligne de Comx
        call omatcopyN(1,no(k),Comx(koc(k)+kloc-1),mo(k),
                                                  owCu(moZ,io(k)+1),not+moZ);
      >;
    >;
  >;
! <b>
  call omatcopy(moU,moU,Domx(kod(u)+kodu(u)-1),moU+mObs,owDu,not+moZ);
>;
```

La partie non marquée par "*" concerne la famille contenant le transfert, avec toutes les copies partielles nécessaire à la modification de structure du système originel, le transfert marqué étant finalement considéré comme le **dernier transfert externe** — on peut rapprocher ce fait de la mise sous Zeus dans **ZOOM** du Borel-transfer.

La première vague d'élimination est particulière dans la famille, dont on présente seuls les calculs y-afférents :

```
do k=1,nof
< if mo(k).gt.0
  < . . .
    < call omatsb2U(mo(k)-1,mo(k)+moU,owDscmx(kowd(k)),mo(k)-1,
                                                  owDscmx(kowd(k)),mo(k)-1);
      call sgesv("in:" mo(k)-1,moZ+no(k),owDscmx(kowd(k)),mo(k)-1,iPiv,
                  "in/out" owDscmx(kowd(k)+(mo(k)-1)*(mo(k)-1)),mo(k)-1,infoa);
! <b> IX,fm,IX  -> Du
! -------------------
call omatsb2U(moZ,moZ,owDu,moZ,owDu,moZ);   *** HORS BOUCLE ***
    < call omatmltad(moZ,mo(k)-1,moZ,Dubuf,moZ,
                   owDscmx(kowd(k)+(mo(k)-1)*(mo(k)-1)),mo(k)-1,owDu,not+moZ);
    >;
! <c> IX,IN,IN  -> Cu
! -------------------
    < call omatmltad(moZ,mo(k)-1,no(k),Dubuf,moZ,
      owDscmx(kowd(k)+(mo(k)-1)*(mo(k)+moU)),mo(k)-1,owCu(1,io(k)+1),not+moZ);
    >;
! <d> Cl,IN,ClX -> Bu
! -----------------
    < call omatmltsb(no(k),mo(k),moU,Bbuf,no(k),
        owDscmx(kowd(k)+(mo(k)-1)*(mo(k)-1)),mo(k)-1,owBu(io(k)+1,1),not+moZ);
    >;
! <e> Cl,IN,IN  -> bspha
! --------------------
    < call omatmltsb(no(k),mo(k)-1,no(k),Bbuf,no(k),
                           owDscmx(kowd(k)+(mo(k)-1)*(mo(k)+moU)),mo(k)-1,
                                              bspha(io(k)+1,io(k)+1),not+moZ);
    >;
```

Par rapport au standard, il y a réduction de la matrice descente, et utilisation des matrices buffers.

Il reste une dernière élimination, dans laquelle les fondus de Fortran apprécieront le subtil jeu d'indices et dimensions ...

```
! Elimination partielle sous U pour laisser d_z, <c_z| et |b_z>
! <f> [I-Du...]m1*Cu
call sgesv("in:" moU,not+1,owDu,not+moZ,iPiv, "out"owCu(1,0),not+moZ,infoa);
;
! <g> IX,u,IX -> bspha
! ------------------
call omatmltsb(not+1,moU,not+1,owBu(0,1),not+moZ,owCu(1,0),not+moZ,
                                                        bspha(0,0),not+moZ);
```

Il reste alors à calculer les éléments propres et les coefficients du gain, identiquement à la Boreleig 102.

On élimine ensuite les matrices pseudo-jacobiennes réduites pour en tirer **aspha**[1] ce qui permet de vérifier globalement les calculs de **bspha** :

```
do ((i=1,not),j=1,not)
< aspha(i,j) = aspha(i,j) - b_z(i)*c_z(j)/d_z;
>
```

qui effectue $A^{spha} = B^{spha} - \frac{1}{d_z} |\ b_z >< c_z\ |$.

Et enfin, mêmes calculs que la 101 pour déterminer la fonction de l'effet-en-retour ρ.

[1] les matrice $C^{\dagger}$, ont été copiées négativement dans **owCu**.

Index

Table des matières