GETTING STARTED WITH ZOOM

I – PROCESSEUR DE CELLULE : PRINCIPES

1) – Introduction

2) – Calculs effectués par la cellule ; calculs effectués par la navette de ZOOM

3) – Les connexions

4) – Un premier plongeon dans ZOOM

II – PROCESSEURS DE TRANSFERTS : PRINCIPES

1) – Equations des transferts

2) – Calculs effectués par le transfert

3) – Les connexions

III – ÉCRITURE D'UN PROBLÈME (ZINIT) : PRINCIPES ET EXEMPLES SIMPLES

1) – Définition de la structure arborescente ; définition et emboîtement des familles

2) – Création des objets cellules

3) – Création des objets transferts

4) – Connexions entre cellules et transfert

5) – Initialisation de l'état initial et des variables (ou paramètres) des objets

IV – ZINIT, VISU : DAVANTAGE D'AVANTAGES

V – QUINCAILLERIE INFORMATIQUE LIÉE AUX PROCESSEURS

1) – Les blocs de données

2) – Parcours de l'ensemble des connexions d'un processeur

3) – Accès au contenu des blocs de données

4) – Accès aux variables réservées

5) – Variables et paramètres propres à chaque processeur

VI – LES DIFFÉRENTS POINTS D'ENTRÉE D'UN PROCESSEUR

1) – Zinit entry

2) – Initialization of structure parameters entry point

3) – Simulation Entry

4) – State dependent parameters entry (advanced programmer only)

5) – Parameter Entry

VII – CHRONOLOGIE DES PRINCIPALES ÉTAPES DE CALCUL ET D'IMPRESSION DANS ZOOM

VII ALGORITHME DE LA NAVETTE *ZOOM/KER* RELATIF AUX MATRICES D DES TRANSFERTS

I – PROCESSEUR DE CELLULE : PRINCIPES

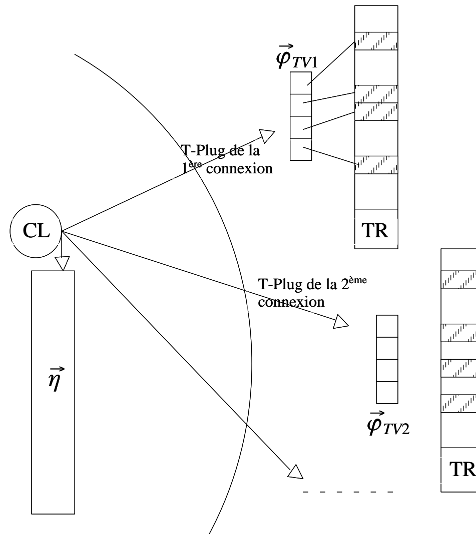
1) – Introduction

Un processeur de cellule est un sous-programme associé au traitement d'un certain type de cellule. Lors de la simulation, le noyau ZOOM effectue une boucle sur toutes les cellules et appelle pour chacune d'elles le processeur associé. Un même processeur peut être appelé pour un grand nombre de cellules, à savoir toutes celles qui sont du même type (c'est-à-dire dont l'évolution relève du même modèle partiel). L'information relative au système est ainsi divisée en deux : d'une part les modèles sont décrits dans les processeurs ; d'autre part, chaque objet relevant de ce modèle (chaque instanciation du modèle) est décrit par un bloc de données, en l'occurrence un bloc de la structure ZEBRA.

La modularité du logiciel se traduit par le fait que c'est le même processeur qui va traiter toutes les instanciations d'un modèle donné.

Ainsi chaque processeur doit pouvoir effectuer le traitement de cellules placées dans des conditions très différentes. Pour comprendre le principe du fonctionnement du processeur, il faut alors imaginer que l'on est à la place de la cellule, ne voyant l'extérieur qu'à travers les filtres offerts par le TEF et la structure informatique de ZOOM : (a) on n'a aucun moyen de voir d'autres variables d'état que celles de la cellule concernée ; (b) de toutes les variables de transfert, on ne voit que celles qui sont connectées à la cellule; (c) de plus, autant par souci d'économie que pour améliorer la souplesse de la structure, un filtre supplémentaire est ajouté par le logiciel ; il s'agit des T-Plugs (que l'on verra en détail plus loin), qui ne laissent voir à la cellule que des variables (bien) choisies dans chaque transfert ; par exemple ceci est utilisé pour ne rendre visibles que les variables de flux d'énergie à une cellule thermique capacitive.

On aboutit ainsi à un schéma que l'on pourrait représenter grossièrement comme ceci :



Alors, partant de l'équation générale d'évolution de la cellule :

$$K\ \partial_t\ \vec{\eta}\ =\ \vec{G}\ (\vec{\eta}\ ,\ \vec{\varphi}_1\ ,\ \ldots\ \vec{\varphi}_{NTV}\ ,\ t) \qquad \text{(I.1)}$$

On obtient une formule discrétisée pour l'évolution dans le temps (on suppose ici, pour simplifier, que l'on travaille uniquement en variation des variables de transfert, et non en variables cumulées) :

$$A\ \delta\ \vec{\eta}\ +\ \sum_{TV=1}^{NTV} B_{TV}\ \delta\ \vec{\varphi}_{TV}\ =\ \vec{\Gamma}\ \delta t \quad \text{soit :} \quad \boxed{\delta\ \vec{\eta}\ +\ \sum_{TV=1}^{NTV} A^{-1}\ B_{TV}\ \delta\ \vec{\varphi}_{TV}\ =\ \delta\ \vec{\eta}_{dec}} \qquad \text{(I.2)}$$

Ainsi, à chaque pas de temps, l'évolution de l'état d'une cellule donnée est caractérisée par tout un ensemble de matrices $(A^{-1}B)_{TV}$, chacune correspondant à la dépendance directe des variables d'état en fonction d'un morceau de vecteur de transfert connecté.
On a (cf. formulaire TEF) :

$$A = K - \frac{\delta t}{2} \frac{\overline{\overline{\partial G}}}{\partial \eta} \qquad \text{(I.3)}$$

$$B_{TV} = - \frac{\delta t}{2} \frac{\overline{\overline{\partial G}}}{\partial \varphi_{TV}} \qquad \text{(I.4)}$$

$$\vec{\Gamma} \delta t = \vec{G}(t) \,.\, \delta t + \frac{\delta t^2}{2} \,.\, \frac{\partial \vec{G}}{\partial t} \simeq \vec{G}\left(t + \frac{\delta t}{2}\right) \delta t \qquad \text{(I.5)}$$

$\delta \vec{\eta}_{dec} = A^{-1} \vec{\Gamma} \delta t$: évolution découplée de la cellule
(celle qui correspondrait à une absence de variation du transfert pendant le pas de temps) (I.6)

Notations :

$\vec{\eta}$: vecteur d'état de la cellule

δt : pas de temps de la simulation à l'instant t (il peut être variable)

$\delta \vec{\eta} = \vec{\eta}(t + \delta t) - \vec{\eta}(t)$

$\vec{\varphi}_{TV}$: ensemble quelconque de composantes du vecteur d'un transfert

$\delta \vec{\varphi}_{TV} = \vec{\varphi}_{TV}(t + \delta t) - \vec{\varphi}_{TV}(t)$

NTV : nombre de vecteurs $\vec{\varphi}_{TV}$ connectés à cette cellule

TV : indice courant de la partie du vecteur de transfert connecté

2) – Calculs effectués par la cellule ; calculs effectués par la navette de ZOOM

Nous sommes à un instant t de la simulation. Le vecteur d'état $\vec{\eta}(t)$ et les vecteurs $\vec{\varphi}_{TV}(t)$ sont définis. Le noyau de ZOOM appelle la partie "simulation" du processeur de chaque cellule.

Processeur de cellule

A partir de $\vec{\eta}(t)$ et de l'ensemble des $\vec{\varphi}_{TV}(t)$, la partie "simulation" du processeur doit calculer :

- les matrices $A^{-1} B_{TV}$ pour l'ensemble des vecteurs $\vec{\varphi}_{TV}$ connectés

- le vecteur $\delta \vec{\eta}_{dec} = A^{-1} \vec{\Gamma} \delta t$

Navette de ZOOM

C'est la navette de résolution (et non le processeur de cellule) qui calcule l'évolution de l'état de la cellule. Après avoir calculé la variation $\delta \vec{\varphi}$ de l'ensemble des transferts, par élimination des variables d'état de l'ensemble des cellules, la navette calcule la variation $\delta \vec{\eta}$ des états entre l'instant t et $t + \delta t$:

$$\delta \vec{\eta} = \delta \vec{\eta}_{dec} - \sum_{TV=1}^{NTV} A^{-1} B_{TV} \delta \vec{\varphi}_{TV}$$

puis remet à jour le vecteur d'état :

$$\vec{\eta}\,(t+\delta\,t) = \vec{\eta}\,(t) + \delta\,\vec{\eta}$$

Remarque : Les sorties à l'instant t seront $\vec{\eta}\,(t)$, $\delta\,\vec{\eta}$, $\delta\,\vec{\eta}_{dec}$

3) – Les connexions

La cellule voit une liste (de 1 à NTV) de vecteurs $\vec{\varphi}_{TV}$, chacun d'eux étant une partie du vecteur $\vec{\varphi}_T$ d'un transfert T . Le vecteur d'état $\vec{\eta}$ de la cellule ne peut dépendre que de lui-même, de cet ensemble de vecteurs $\vec{\varphi}_{TV}$, et éventuellement explicitement du temps (cf. équation I.1).

En fait, rien n'empêche $\vec{\varphi}_{TV}$ d'inclure d'autres variables que celles dont dépend effectivement la cellule. Le processeur de cellule va faire des hypothèses sur la position des variables pertinentes dans les $\vec{\varphi}_{TV}$.

Chaque processeur de cellule n'admet qu'un nombre réduit de type de vecteurs $\vec{\varphi}_{TV}$. Par exemple, pour la cellule capacitive " $ND1$ ", on suppose que la première composante de tous les vecteurs $\vec{\varphi}_{TV}$ est un flux de chaleur et que l'évolution de la température T de la cellule ne dépend que de la somme de ces flux:

$$C\,\frac{dT}{dt} = \sum_{TV=1}^{NTV} \varphi_{TV}\,(1)$$

avec :

> NTV : nombre total de vecteurs $\vec{\varphi}_{TV}$ connectés à cette cellule
>
> $\varphi_{TV}\,(1)$: 1ère composante du vecteur $\vec{\varphi}_{TV}$
>
> C : capacité calorifique

Les autres composantes de $\vec{\varphi}_{TV}$, si elles existent, n'ont aucune influence directe sur la cellule (on a toutefois intérêt à limiter la dimension de $\vec{\varphi}_{TV}$ pour diminuer la taille des matrices); le processeur *ND1* les ignorera.

Lors de la définition et l'initialisation du problème à traiter (dans ZINIT), l'utilisateur doit ainsi définir les vecteurs $\vec{\varphi}_{TV}$ à partir des vecteurs $\vec{\varphi}$ des transferts (instruction **`T-Plug`** du ZDL) et en fonction de l'exigence des cellules.

De façon "standard", une cellule n'a accès qu'à la "nature physique" des transferts connectés ainsi qu'à la dimension des vecteurs $\vec{\varphi}_{TV}$. La valeur de l'indice TV n'est pas significative. (Ils conservent cependant l'ordre de la numérotation des **`T-Plug`** dans zinit) .

Ainsi, l'allure générale du cœur d'un processeur de cellule, dans la partie "simulation", est la suivante:

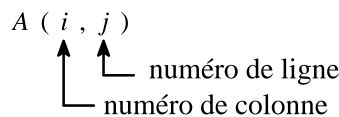
```
boucle sur l'ensemble des vecteurs φ⃗_TV
<   si (Nature-du-transfert = ceci)
    <   utilisation de φ⃗_TV ;
        calcul de la contribution éventuelle à δ η⃗_dec
        calcul de A^-1 B_TV ;
        ...
    >
    si (Nature-du-transfert = cela)
```

```
        <   utilisation de  φ⃗_TV ;
            calcul de la contribution éventuelle à  δ η⃗_dec
            calcul de  A^-1 B_TV ;
        >
        etc.
    >   "fin de boucle"
    calcul final de  δ η⃗_dec
```

4) – Un premier plongeon dans le langage des processeurs de ZOOM

4.1) – Nom de variables

• Matrice : pour les matrices, la convention (convention du CERN) est que le premier indice est celui des colonnes et le second celui des lignes.



•Variables réservées, propres aux cellules :

ETA : $\vec{\eta}$

GAMMA : $\delta\ \eta_{dec} = A^{-1}\ \vec{\Gamma}\ \delta t$

FI_TV : $\vec{\varphi}_{TV}$

ABt_TV `( i , j )` : matrice transposée de $A^{-1}\ B_{TV}$

avec :

i : numéro de colonne de la matrice transposée de $A^{-1}\ B_{TV}$, qui correspond donc à la i **ème composante du vecteur d'état** $\vec{\eta}$

j : numéro de ligne de la matrice transposée de $A^{-1}\ B_{TV}$, qui correspond donc à la j **ème composante du vecteur** $\vec{\varphi}_{TV}$

• Autres variables réservées :

DTIME : $\delta\ t$

TIME : t

ZPrint : variable logique, pilotée par l'instruction "`Print out every N steps`" dans ZINIT ; vrai si l'on doit imprimer.

• Convention de signe

Pour les cellules, la convention de signe est que l'on compte positivement les flux entrants (convention thermodynamique).

Pour les transferts à plus de deux extrémités (transfert radiatif par exemple), les flux seront calculés de manière à respecter cette convention de signe pour chacune des extrémités.

Pour les transferts à deux extrémités par contre, on a généralement une même variable de flux pour les deux extrémités, sortant d'une cellule, entrant dans l'autre : l'une des cellules devra prendre directement cette variable ; l'autre cellule devra changer le signe de cette variable. La fonction **SIGN_TV** permet de gérer ceci et prend les valeurs + 1 ou – 1 selon l'extrémité du transfert à laquelle est connectée la cellule.

Par exemple, prenons le cas d'un processeur de type "conductif" ("CD1"), dans lequel la conduction est modélisée par une conductance H :

$$Flux = H\ (T_1 - T_2)$$

Le flux est orienté positivement de 1 vers 2. Avec cette convention de signe, la cellule 1 doit changer le signe du flux ainsi calculé ; la cellule 2, non.

Remarque : Pour qu'à la fois les cellules et les transferts puissent reconnaître les extrémités 1 et 2, la convention suivante est souvent adoptée : on crée deux `T-Plug` (numérotés 1 et 2) ; on connecte chacun de ces `T-Plug` à une (ou des) variables d'état d'une cellule (en réalité, on les connecte à un `C-Plug`, lui-même pointant sur une (ou des) variables d'état de la cellule). Les `T-Plug` étant ordonnés, le transfert peut savoir quelle variable d'état correspond à l'extrémité 1 et laquelle correspond à l'extrémité 2 ; de plus, chacune des cellules peut savoir à quelle extrémité elle est connectée. La fonction **SIGN_TV**, appelée d'une cellule pour une connexion donnée, renvoie la valeur – 1 ou + 1 suivant que la connexion est établie sur l'extrémité 1 ou l'extrémité 2. Lorsque le transfert n'a pas deux extrémités, cette fonction prend toujours la valeur 1.

4.2) – Exemple avec un seul type de connexion : cellule capacitive ND1

- Hypothèse sur les connexions : pour tous les vecteurs $\vec{\varphi}_{TV}$ connectés à cette cellule, la 1ère composante de ce vecteur $\left(\varphi_{TV}\ (1) \right)$ est un flux dont le signe est déterminé par la fonction SIGN–TV.
- Variable d'état : une seule ; T
- Equation d'état : $$C \frac{dT}{dt} = G\ (\ \vec{\varphi}_1\ ,\ \vec{\varphi}_2\ ,\ \dots\ ) = \sum_{TV=1}^{NTV} \varphi_{TV}\ (1)$$
- Equation discrétisée en temps : $\delta\ T + \sum_{TV=1}^{NTV} A^{-1}\ B_{TV}\ \delta\ \varphi_{TV}\ (1) = \delta\ T_{dec}$

avec :

$$A = C$$

$$B_{TV} = -\frac{\delta t}{2} \frac{\partial G}{\partial \varphi_{TV}\ (1)} = -\frac{\delta\ t}{2}$$

$$\delta\ T_{dec} = A^{-1}\ \Gamma\ \delta\ t$$

$$\Gamma\ \delta\ t = G(t)\ .\ \delta\ t = \sum_{TV=1}^{NTV} \varphi_{TV}\ (1)\ .\ \delta\ t$$ (la cellule n'a pas de terme source intrinsèque)

- Variables à calculer :

$$(A^{-1}B_{TV})^T\ :\ \mathbf{ABt_TV}\ (1,1) = -\frac{\delta\ t}{2} * \frac{SIGN_TV}{C}$$

$$\delta T_{dec}\ :\ \mathbf{GAMMA}(1) = \frac{\delta\ t}{C} \sum_{TV=1}^{NTV} SIGN_TV * FI_TV\ (1)$$

• Ecriture en méta-langage du cœur de la partie ”simulation” de ce processeur

```
! ================================================================
! ==   Capacitance Processor   ;   Cell processor, type = ND1;   ==
! === Cell parameters ===
    Heat_capa : real ;
! ==== Simulation Entry Point
! ==========================================================
! == SumFiT ,  Dtime_2 , Dtime_1 : FORTRAN local variables
  Dtime_2 = .5*DTIME;      Dtime_1 = DTIME;
  SumFiT=0.;  "Sum of Heat Flux "
  Loop_on_all_connected Fi_TV vectors  (TV = 1,NTV)
  <
    if (dimension of FI_TV <> 0)       "détection des T-probes"
    <
      SumFiT = SumFiT + Sign_TV * FI_TV(1);
      ABt_TV(1,1) = -Sign_TV * Dtime_2 / Heat_capa ;
    > "end if"
  > "end loop"
  Gamma(1) = SumFiT * Dtime_1 / Heat_Capa ;
RETURN;
! ==========================================================
```

4.3) – Exemple avec trois types de connexions : cellule correspondant à un volume fini d’air humide jamais saturé

• Hypothèse sur les connexions : les transferts correspondant à 3 “natures physiques” différentes peuvent être connectés :

- Les transferts de nature “flux de chaleur” ; la 1ère composante de $\vec{\varphi}_{TV}$ doit être le flux thermique.
- Les transferts de nature “flux d’humidité” ; la 1ère composante de $\vec{\varphi}_{TV}$ doit être le flux de masse.
- Les transferts de nature “flux de chaleur et d’humidité” ; la 1ère composante de $\vec{\varphi}_{TV}$ est le flux thermique, la 2ème le flux de masse.

• Deux variables d’état :

T : température (K)

W_m : teneur en eau (kg d’eau par kg d’air sec)

• Deux types de variables de transferts connectés :

φ_T : flux thermique (W/m^2)

φ_{W_m} : flux de masse de vapeur d’eau (en kg/m^2/s)

• Equations d’état lorsque l’air n’est pas saturé en humidité

$$\boxed{C_T \, . \, \frac{\partial T}{\partial t} = \Sigma \, \varphi_T} \qquad C_T = \varrho \, . \, C_p \, . \, \Delta z \quad \text{(surface unité)}$$

$$\boxed{C_W \, . \, \frac{\partial W_m}{\partial t} = \Sigma \, \varphi_{W_m}} \qquad C_W = \varrho \, . \, \Delta z$$

ϱ : masse volumique

Δz : épaisseur de la tranche d'air

C_p : chaleur massique

- La discrétisation temporelle est identique à celle de l'exemple précédent ; les variables à calculer sont de la même forme. L'unique différence par rapport à l'exemple précédent provient des différentes connexions possibles.
- Ecriture en méta-langage du cœur de la partie "simulation" de ce processeur :

```
! ================================================================
! *     Humid Air Processor
! ================================================================
!  eta(1) : T (K)
!  eta(2) : Wm (kg of water/kg of air)
!  === Cell parameters ===
    Heat_capa    , Water_capa     :real ;
! ==== Simulation Entry Point
! ===============================================================
! == SumFiT , SumFiWm ,  Dtime_2 , Dtime_1 : FORTRAN local variables
  Dtime_2 = .5*DTIME;  Dtime_1 = DTIME;
  SumFiWm=0.;   SumFiT=0.;
! ******* si air NON sature en humidite *********
  Loop_on_all_connected Fi_TV vectors  (TV = 1,NTV)
  < if  (Nature_of_Transfert  = flux_thermique)
    < SumFiT = SumFiT + Sign_TV * FI_TV(1);
      ABt_TV(1,1) = -Dtime_2 * Sign_TV / Heat_capa ;
    >ElseIf  (Nature_of_Transfert  = flux_d'humidite)
    < SumFiWm = SumFiWm + Sign_TV * FI_TV(1);
      ABt_TV(2,1) = -Dtime_2 * Sign_TV / Water_capa ;
    >ElseIf  (Nature_of_Transfert  =  flux de  chaleur  et  d'humidite)
    < SumFiT = SumFiT + Sign_TV * FI_TV(1);
      SumFiWm = SumFiWm + Sign_TV * FI_TV(2);
      ABt_TV(1,1) = -Dtime_2 * Sign_TV / Heat_capa ;
      ABt_TV(1,2) = 0.;
      ABt_TV(2,2) = -Dtime_2 * Sign_TV / Water_capa ;
      ABt_TV(2,1) = 0.;
    > "end if"
  > "end loop"
  Gamma(1) = Dtime_1 * SumFiT / Heat_capa ;
  Gamma(2) = Dtime_1 * SumFiWm / Water_capa ;
!
RETURN;
!==================================================
```

- Remarque : Sign_TV est une fonction qui prend une valeur différente dans chaque cas, suivant l'environnement des connexions

II – PROCESSEURS DE TRANSFERTS : PRINCIPES

1) – Equations des transferts

De façon générale, les variables d'un transfert peuvent dépendre des variables d'état des cellules auxquelles ce transfert est connecté, des variables des transferts connectés à ces mêmes cellules, et éventuellement explicitement du temps. Leur équation a la forme d'une équation de contrainte :

$$\vec{\varphi} = f(\vec{\eta}_1, \ldots, \vec{\eta}_{NSV}, \vec{\varphi}_1, \ldots, \vec{\varphi}_N, t)$$

avec : $\vec{\varphi}$: vecteur du transfert étudié

$\vec{\eta}_1, \ldots, \vec{\eta}_{NSV}$: ensemble des parties des vecteurs d'état des cellules connectées

$\vec{\varphi}_1, \ldots, \vec{\varphi}_N$: ensemble des parties des transferts connectés à l'ensemble des cellules connectées au transfert étudié

Dans un premier temps, pour simplifier, nous traiterons des transferts dépendant uniquement des états des cellules connectées, appelés généralement "transferts sans matrice D". L'équation des variables de transfert est alors la suivante :

$$\vec{\varphi} = f(\vec{\eta}_1, \ldots, \vec{\eta}_{NSV}, t)$$

On obtient une formule discrétisée pour l'évolution en temps :

$$\delta t \sum_{SV=1}^{NSV} C_{SV}^{t} \; \delta \, \vec{\eta}_{SV} - \delta \, \vec{\varphi} = \vec{\Omega} \; \delta t \qquad \text{(I.7)}$$

avec : . $\vec{\varphi}$: vecteur du transfert

$\delta \; t$: pas de temps

$\delta\vec{\varphi} = \vec{\varphi}(t + \delta t) - \varphi(t)$

$\vec{\eta}_{SV}$: partie quelconque du vecteur d'état d'une cellule

$\delta \, \vec{\eta}_{SV} = \vec{\eta}_{SV}(t + \delta t) - \vec{\eta}_{SV}(t)$

NSV : nombre de vecteurs $\vec{\eta}_{SV}$ connectés à ce transfert

SV : indice courant de la partie du vecteur d'état connecté

$$C_{SV}^{T} \; \delta t = \frac{\partial \overline{\overline{f}}}{\partial \eta_{SV}} \qquad \text{(I.8)}$$

$$\vec{\Omega} \; \delta t = -\frac{\partial \vec{f}}{\partial t} \; . \; \delta t \qquad \text{(I.9)}$$

Ces deux expressions ne sont valables que lorsque l'on travaille en variation des variables de transfert et non en variables cumulées (cf. formulaire TEF). On notera d'ailleurs que les différents coefficients $\delta \; t$, qui peuvent sembler bizarres, proviennent de ce que ces formules ont été initialement écrites pour ce dernier cas.

2) – Calculs effectués par le transfert

Nous sommes à un instant t. L'ensemble des vecteurs $\vec{\eta}_{SV}\ (t)$ est défini. Le noyau de ZOOM appelle la partie "simulation" du processeur de chaque transfert, qui doit calculer :

- $\vec{\varphi}\ (t) = f\ (\vec{\eta}_1, \ldots, \vec{\eta}_{NSV})$

- les matrices $C_{SV}^T\ .\ \delta t$ pour l'ensemble des vecteurs $\vec{\eta}_{SV}$ connectés

- le vecteur $\vec{\Omega}\ .\ \delta t$

Les variables réservées correspondantes sont les suivantes :

`FI` : $\vec{\varphi}$

`OMEGA` : $-\vec{\Omega}\ \delta t = \frac{\partial \vec{f}}{\partial t}\ \delta t$

`ETA_SV` : $\vec{\eta}_{SV}$

`Ct_SV (i,j)` : élément de $C_{SV}^T\ \delta t = \frac{\partial \overline{\overline{f}}}{\partial \eta_{SV}}$

avec : `i` : numéro de colonne de la matrice C^T, qui correspond donc à la **i^ème^ composante du vecteur** $\vec{\eta}_{SV}$

`j` : numéro de ligne de la matrice C^T, qui correspond donc à la **j^ème^ composante du vecteur** $\vec{\varphi}$

3) – Les connexions

La problématique est la même que pour les cellules.

Le transfert voit une liste (de 1 à NSV) de vecteurs $\vec{\eta}_{SV}$, chacun d'eux étant une partition du vecteur d'état $\vec{\eta}$. Un processeur de transfert n'admet qu'un nombre limité de type de vecteurs $\vec{\eta}_{SV}$.

Par exemple, pour le processeur de transfert "CD1", de type conductif (résistance thermique), on suppose que la première composante des $\vec{\eta}_{SV}$ connectés est obligatoirement une température.

Dans ZINIT, l'utilisateur doit définir ces vecteurs $\vec{\eta}_{SV}$ à partir du vecteur $\vec{\eta}$ de chaque cellule ; ceci est réalisé à l'aide de l'instruction **`C-Plug`** du ZDL (cf. § III.2).

De façon "standard", un transfert a accès à la "nature physique" des cellules connectées, à la dimension du vecteur $\vec{\eta}_{SV}$. L'indice SV lui-même n'est pas significatif ; par contre on connaît le numéro du `T-Plug` auquel ce vecteur $\vec{\eta}_{SV}$ est connecté [1].

Cette connaissance du numéro du `T-Plug` auquel un vecteur $\vec{\eta}_{SV}$ est connecté permet de traiter les problèmes d'orientation souvent rencontrés dans les processeurs de transfert.

Reprenons l'exemple esquissé (§ I.3) du transfert conductif "CD1" (conductance thermique). Son équation est :

$$Flux = H * \left(T(1) - T(2)\right)$$

$$C_{SV}^T\ .\ \delta t = \frac{\partial \overline{\overline{f}}}{\partial \eta_{SV}} = \pm\ H$$

$$\Omega\ \delta t = -\frac{\partial f}{\partial t}\ .\ \delta t = 0 \qquad \text{(pas de terme source)}$$

1. Plus précisément, les `T-Plug` sont rangés dans l'ordre des numéros croissants.

Hypothèses de connexions :

- uniquement deux vecteurs $\vec{\eta}_{SV}$ sont connectés à ce transfert

- la 1ère composante de chacun de ces deux vecteurs $\vec{\eta}_{SV}$ est une température

- chacun de ces deux vecteurs $\vec{\eta}_{SV}$ est connecté à un T–Plug différent.

L'accès aux variables $T(1)$ et $T(2)$ est ainsi totalement défini :

- si le vecteur $\vec{\eta}_{SV}$ est connecté au 1er T–Plug, alors $T(1) = \eta_{SV}(1)$

- si le vecteur $\vec{\eta}_{SV}$ est connecté au 2ème T–Plug, alors $T(2) = \eta_{SV}(1)$

L'écriture en méta-langage de la partie “simulation” de ce processeur est la suivante :

```
  SUBROUTINE TCD1(LTR);
!===============================================================
!====  parameters ====
    Conduct : real ;  " conductance"
!==== Simulation Entry Point
!==============================================================
! ==  T(1) and T(2) :  FORTRAN locale variables
  Loop_on_all ETA_SV vectors
  < if (First_Tplug)
    < T(1) = Eta_SV(1);    TSIGN = 1.;
    >else if (Second_T-Plug)
    < T(2) = Eta_SV(1);    TSIGN = -1.;
    >
    Ct_SV(1,1) = Conduct * TSIGN ;
  >
  FI(1) = Conduct*(T_Edge(1)-T_Edge(2));
  Omega(1) = 0. ;
RETURN;
!======================================================
```

III – ÉCRITURE D'UN PROBLÈME (ZINIT) : PRINCIPES ET EXEMPLES SIMPLES

La phase d'initialisation d'un problème dans ZOOM a plusieurs objectifs :

- définir la structure arborescente d'analyse
- créer les objets cellules et transferts (instanciation des processeurs)
- établir les connexions entre cellules et transferts
- initialiser les variables d'état et d'autres variables (ou paramètres) des différents objets.

1) – Définition de la structure arborescente ; définition et emboîtement des familles

- Rites : avant toute chose, création de l'objet structurant le problème (ZEUS) et de l'univers étudié (UNIVERSE):

```
create ZEUS ;
create UNIVERSE ;
```

Après ces saintes (et obligatoires) écritures, on s'adressera à ces deux fantômes via leur petit nom :

Z pour ZEUS

U pour UNIVERSE

- Création d'une famille :

create FM `Fm_Name` ; ou **create FM** `Fm_Name-Num` ;

avec : `Fm_Name` : nom de la famille (au plus 4 lettres)
`Num` (optionnel) : numéro de la famille (permet de créer N familles du même nom)

Exemple :

```
create FM DMON ;         "un demon"
do I = 1, N
< create FM STFM-I ;     "et N Ste Famille"
>                        "sont dans un bateau"
"Nota Bene : les commentaires sont entre doubles quotes"
! ou après un point d'exclamation en première colonne
```

- Déplacement dans l'arbre ; définition de la famille courante :

Après la création de ZEUS et UNIVERSE, la famille courante est l'Univers (U).

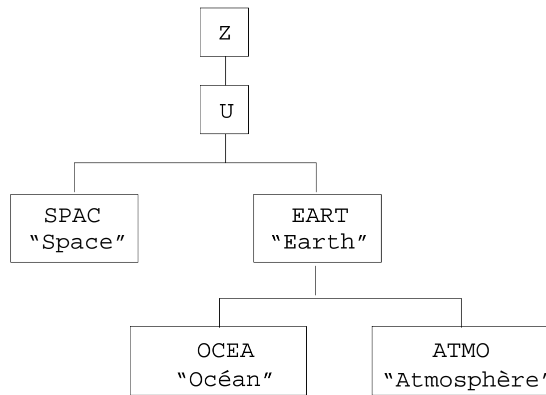
Après avoir créé une famille `FM1` sous la famille courante, on va dans cette famille (on la définit comme famille courante) par l'instruction :

```
under FM1
<
>
```

Cette instruction utilise la notion de "bloc MORTRAN" (ouvert par < et fermé par >) : après l'instruction under FM1 < on est dans la famille FM1 ; après la fermeture du bloc (>), on revient à la famille courante précédente (d'avant le `under FM1 <`).

Cette instruction est emboîtable ; elle permet ainsi de créer des familles emboîtées et de parcourir l'arbre.

Exemple : On veut créer la structure correspondant à l'arbre ci-dessous :




```
create ZEUS ;
create UNIVERSE ;
"on est sous l'Univers"
create SPAC ;
create EART ;
under SPAC
< ....
  "on est dans SPAC"
> "on revient dans U"
under EART
< "on est dans EART"
  create OCEA ; create ATMO ;
  under OCEA
  < "on est dans OCEA"
    ....
  >
  "on est dans EART"
  under ATMO
  < "on est dans ATMO"
    ....
  > "on revient dans EART"
> "on revient dans U"
```

2) – Création des objets cellules

- Création :

  create CL `Cl_Name {-Num}`, **type =** `Cl_Proc_Name {Arg}` ;
  { } voulant dire optionnel

avec :

`Cl_Name` : nom de la cellule (maximum 4 caractères)

`Num` : (optionnel) numéro (entier positif)

`Cl_Proc_Name` : nom du processeur de cellule (maximum 7 caractères)

`Arg` : (optionnel) argument entier utilisé par certains processeurs (voir ?)

Après sa création, une cellule est identifiée par son nom complété, le cas échéant, par un numéro.

- Définition des parties du vecteur d'état visible de l'extérieur

Nous avons vu précédemment (§ II) que les transferts ne voient pas directement le vecteur d'état de chacune des cellules auxquelles ils sont connectés ; ils ne voient qu'une partie de chacun de ces vecteurs d'état (que nous avons notés $\vec{\eta}_{SV}$).

Ainsi l'utilisateur définit un (ou plusieurs) "sous-vecteur d'état" $\vec{\eta}_{SV}$ pour chaque objet cellule, en fonction des exigences du processeur de transfert, à l'aide de l'instruction :

```
C-Plug Plug_Num : St_Var_List ;
```

avec :

`Plug_Num` : numéro du C–Plug de cette cellule (entier naturel non nul). Ce numéro est utilisé dans ZINIT pour la connexion avec les transferts, mais n'est actuellement pas visible par les processeurs eux-mêmes

`St_Var_List` : Liste des numéros de composantes du vecteur d'état, séparé par des virgules, définissant le vecteur $\vec{\eta}_{SV}$

Exemple :

```
C-Plug 1 : 1, 5, 2 ;
```

définit un vecteur $\vec{\eta}_{SV}$ à 3 composantes (en notant $\vec{\eta}$ le vecteur d'état de la cellule) :

$\eta_{SV}(1) = \eta(1)$
$\eta_{SV}(2) = \eta(5)$
$\eta_{SV}(3) = \eta(2)$

```
C-Plug 2 : 1, 4 ;
```

définit un second vecteur $\vec{\eta}_{SV}$ à 2 composantes :

$\eta_{SV}(1) = \eta(1)$
$\eta_{SV}(2) = \eta(4)$

- Alias

On ne peut accéder directement à une cellule via son nom, que si l'on est dans la famille à laquelle elle appartient. Pour pouvoir accéder à une cellule de n'importe où, on crée l'équivalent d'un "alias" à l'aide de l'instruction :

```
Set Alias_Name = Cl_Name ;
```

avec : `Alias_Name` : nom de l'alias (maximum 4 caractères)

Le nom de l'"alias" peut être (et est en général, lorsque toutes les cellules ont un nom différent) identique à celui de la cellule. L'alias peut être dimensionné :

`Set Alias_Name {(i)} = Cl_Name {- j} ;` {} voulant dire optionnel

avec : `i` : entier strictement positif

Ceci est couramment utilisé lorsque les objets cellules sont définis avec un nom et un numéro.

Exemple :

```
do i = 1, N
< create CL Mesh-i, type = .... ;
  set Mesh (i) = Mesh-i ;
>
```

Remarque : les "alias" peuvent également être utilisés avec des familles.

- Pour jouer

Geler une cellule : on crée un objet cellule, mais on veut que cette cellule n'évolue pas en fonction du temps (quel que soit le type de processeur). On peut faire ceci de deux façons différentes :

- lors de la création de la cellule :

```
create N-Cl Cl_Name {-Num}, type = .... ;
```

- ou, après avoir créé la cellule :

```
Freeze Cl_Name {-Num} ;
```

Déplacer une cellule dans une autre famille :

Après avoir créé une cellule dans une famille donnée, on peut la déplacer et la mettre dans une autre famille :

.... **MOVE** `Cl_Name` **TO** `Fm_Name ;`

avec : .... `Cl_Name` : nom de la cellule ; si on a créé un alias pour cette cellule (SET ...), on peut l'utiliser
.... `Fm_Name` : nom de la cellule ; si on a créé un alias pour cette famille, on peut l'utiliser.

Attention ! : Si on n'a pas fait d'alias, il faut que les deux objets (cellule et famille) soient dans la famille d'où l'on fait le `MOVE`. Il va sans dire que les alias sont donc très utiles pour cette instruction.

3) – Création des objets transferts

• Création :

create TR, type = `Tr_Proc_Name {Arg}` ; {} voulant dire optionnel

avec : `Tr_Proc_Name` : nom du processeur de transfert
`Arg` : (optionnel) argument entier utilisé par certains processeurs (voir ?)

Contrairement aux cellules, les transferts n'ont pas de nom d'objet. On ne peut ainsi accéder et faire des opérations relatives à un transfert (T–Plug, connexions avec les cellules, valeurs des paramètres du transfert ...) qu'immédiatement après sa création.

• Définition des parties du vecteur de transfert visibles de l'extérieur

La démarche est la même que pour les cellules (cf. III.2).

Nous avons vu (§ I) que les cellules ne voient pas directement le vecteur de transfert de chacun des transferts auxquels elle est connectée ; elle n'en voit qu'une partie, notée $\vec{\varphi}_{TV}$. L'utilisateur définit ces "sous-vecteurs $\vec{\varphi}_{TV}$ " pour chaque objet transfert, en fonction des exigences des processeurs de cellule et de transfert.

T-Plug `Plug_Num : Tr_Var_list ;`

avec :

`Plug_Num` :	numéro du T–Plug du transfert
`Tr_Var_List` :	Liste des numéros de composantes du vecteur de transfert, séparé par une virgule, définissant le vecteur $\vec{\varphi}_{TV}$

Exemple :

`T-Plug 1 : 5, 7, 2, 3 ;` définit $\vec{\varphi}_{TV}$ tel que
(en notant $\vec{\varphi}$ le vecteur du transfert)

$$\varphi_{TV}(1) = \varphi(5)$$
$$\varphi_{TV}(2) = \varphi(7)$$
$$\varphi_{TV}(3) = \varphi(2)$$
$$\varphi_{TV}(4) = \varphi(3)$$

Ordre des Plug :

Vu d'un processeur de transfert, les T-Plug sont rangés dans l'ordre des numéros croissants.

• Divers et variés

T–Probe : lorsqu'un transfert dépend d'une cellule, mais que celle-ci ne dépend pas du transfert, on peut créer un `T-Probe.`

T-Probe : Plug_Num ;

Ceci crée l'équivalent d'un T–Plug, mais pour lequel la dimension de $\vec{\varphi}_{TV}$ est nulle. ZOOM ne crée pas de matrice $A^{-1}\ B_{TV}$ associée à cette connexion.

Geler un transfert

On crée un transfert dont le vecteur restera constant dans le temps et égal à sa valeur initiale.

`create` **N-**`TR, type = ... ;`

La valeur initiale du vecteur de transfert est, selon les processeurs, soit calculée dans le “S entry” (cf. § VI.2), soit initialisée dans ZINIT.

Forcer la position d’un transfert dans l’arbre des familles

- Notion de position “naturelle” d’un transfert
- Lorsque l’on crée un transfert dans ZINIT, sa position dans l’arbre n’est pas définie. C’est uniquement après avoir créé l’ensemble de l’arbre que les transferts sont placés par “OFRAME” (cf. § VIII) à leur position naturelle. On peut toutefois obliger un processeur de transfert à être dans une famille **au-dessus** de sa position “naturelle”.

Il faut d’abord se placer dans la famille où l’on veut forcer le transfert à être (under ... <), puis créer le transfert avec l’option “–F” :

```
create-F TR, type = ... ;
```

Pour positionner un transfert sous ZEUS (pour faire de l’analyse de couplage par exemple) ou sans l’UNIVERSE, on peut simplement faire, où que l’on soit dans l’arbre :

```
create-Z TR, type = ... ;

create-U Tr, type = ... ;
```

4) – Connexions entre cellules et transfert

On a créé des `C-Plug` pour les cellules, et des `T-Plug` pour les transferts. Il reste à établir une connexion, un lien, entre `C-Plug` et `T-Plug` :

```
JOIN T_Plug_Num TO (Cl_Name,C_Plug_Num) {,(Cl_Name_2,C_Plug_Num)} ;
```

avec :

`T_Plug_Num` : numéro du `T-Plug` à connecter

`Cl_Name_1` : nom de la cellule sur laquelle on établit la connexion. Comme d’habitude, ce nom est soit le nom de la cellule (si elle est dans la famille courante), soit le nom de l’alias

`C_Plug_Num` : numéro du `C-Plug` de la cellule sur laquelle on établit la connexion

5) – Initialisation de l’état initial et des variables (ou paramètres) des objets

Pour les objets cellules, il faut initialiser la valeur initiale du vecteur d’état, l’équation d’une cellule étant une équation d’évolution.

Pour les objets transferts, sans matrice D, c’est-à-dire pour des transferts qui ne dépendent que des états des cellules auxquelles ils sont connectés (cf. § II.1), il est inutile d’initialiser le vecteur de transfert : il sera calculé directement par ZOOM.

Pour les objets cellules, on peut soit initialiser l’état et les autres variables juste après la création de la cellule, soit (ce qui est le plus courant) regrouper ces initialisations à la fin de ZINIT, après avoir fait un

```
GET Cl_Name;
```

pour chacune des cellules.

Par contre, les transferts n’étant pas individualisés par un nom, il faudra initialiser les variables juste après la création de chacun d’eux.

- Initialisation des variables d’état d’une cellule

Ceci se fait à l’aide de l’instruction :

```
fill_ETA
< [i] = expression ;
   ....
>
```

avec :

i : numéro de la composante du vecteur d'état d'une cellule

expression : expression arithmétique quelconque

A l'intérieur du "bloc MORTRAN" `fill_ETA <    >` , on peut faire des boucles, des tests, etc.

Pour initialiser tout le vecteur d'état à une même valeur, on peut faire :

```
fill_ETA < [.] = Valeur ;   >
```

Exemple :

On veut initialiser un vecteur d'état à 5 composantes. La première composante du vecteur d'état vaut 5, la seconde la valeur de la variable FORTRAN X (initialiser avant) et les 3 dernières la variable FORTRAN Y.

```
fill_ETA
< [1] = 5 ;
   [2] = X ;
   do I = 3,5 < [I] = Y ; >
>
```

- Initialisation des variables (ou paramètres) d'une cellule

Un processeur de cellule possède un bloc de données (appelé CD pour Cell Description block) contenant un certain nombre de variables. Ce bloc est déclaré au début de chaque processeur ; les variables appartenant à ce bloc ont un nom symbolique, un type, et éventuellement une dimension.

Suivant les processeurs, un certain nombre de variables doivent être initialisées depuis ZINIT (des commentaires au début de chaque processeur doivent indiquer les variables à initialiser) à l'aide de l'instruction :

`fill_CD`

`< [Var_Name] : type = expression ;` "pour les variables non dimensionnées"

`[Var_Name (i{,j})] : type = expression ;` "pour les variables dimensionnées"

`>`

avec :

`Var_Name` : nom de la variable tel qu'il a été défini dans le bloc de données CD en tête du processeur de cellule

`type` : type de la variable : `i` pour intégrer et `r` pour real

`i` et `j` (le cas échéant) : indices

Exemple :

```
fill_CD
< [Heat_capa]:r = CpSol;
  [Water_capa]:r = WmCpsol;
  [RoAirSec]:r =  RoCtr(1);
  [Sature]:i = 2 "tjrs sature" ;
>
```

- Initialisation des variables (ou paramètres) d'un transfert

Exactement pareil que précédemment, mais en remplaçant `fill_CD` par **`fill_TD`**

• Initialisation des variables du vecteur de transfert

Ceci est inutile pour les exemples simples, classiques.

Si cela est nécessaire, la syntaxe est strictement identique à celle des cellules, mais en remplaçant `fill_ETA` par **`fill_FI`**.

IV – ZINIT, VISU : DAVANTAGE D'AVANTAGES

V – QUINCAILLERIE INFORMATIQUE LIÉE AUX PROCESSEURS

1) – Les blocs de données

A un processeur correspond une seule routine. A chaque création d'objet (cellule ou transfert), on crée et remplit des blocs de données contenant toute l'information pour que les processeurs puissent faire le traitement spécifique à chacun de ces objets.

La gestion de ces blocs de données est donc une partie essentielle de ZOOM. Elle est basée sur le logiciel ZEBRA, développé par le CERN, qui permet la gestion arborescente et dynamique de blocs de données. Nous avons développé un langage (Z.B.M., Zoom Bank Manager) qui permet de manipuler et d'accéder à ces blocs de données de façon relativement transparente pour l'utilisateur.

Ces différents blocs de données sont chaînés entre eux. La liste des chaînages nécessaires à un processeur doit être déclarée, au début, par l'instruction **Local _valid _chains** (voir ZBM Manual). Il existe une liste type pour les processeurs de cellule, les processeurs de transfert sans matrice D, les processeurs de transfert avec matrice D (voir le modèle de ces différents types de processeurs).

Il faut ensuite donner une "racine" permettant d'accéder aux blocs de données chaînés. Pour un transfert, cela est effectué par l'instruction **from TR (LTR) <** , pour une cellule par **from CL (LCL) <** . Ces instructions (ces "blocs MORTRAN") sont fermés par > tout à la fin du processeur.

2) – Parcours de l'ensemble des connexions d'un processeur

Pour les cellules :

Précédemment (§ I), pour parcourir l'ensemble des vecteurs $\vec{\varphi}_{TV}$ connectés à une cellule, nous avions écrit la boucle en méta-langage :

```
Loop-on-all-connected Fi-TV vectors
<
> "end loop"
```

Cette boucle s'écrit en ZBM.

```
  skip thru SV(LSV)   "boucle sur les C-Plug de la cellule"
  < skip thru CX(LCX) "Boucle sur l'ensemble des connex. de chaque C-Plug"
    < get TV(LTV);    "On arrive au T-Plug du transfert connecté"
      if ((LTV).Nb_Tr_Var.EQ.0) to next LCX; "Si la dimension de FI_TV"
                          "est nulle, on passe a la connexion suivante"
      ...... (calcul de ABt_TV) .....
    > "end skip thru CX"
  > "end skip thru SV"
```

Pour les transferts sans matrice D

Précédemment (§ II), pour parcourir l'ensemble des vecteurs $\vec{\eta}_{SV}$ connectés à un transfert, nous avions écrit la boucle suivante :

```
Loop-on-all-connected Eta-SV vectors
<
> "end loop"
```

Cette boucle s'écrit en ZBM.

```
 skip thru TV(LTV)     "boucle sur les T-Plug du transfert"
  <  NCX=(LTV).Nb_Cx_CX;
     read thru CX(LCX) (J=1,NCX) "Boucle sur l'ensemble des connexions de"
     <                           "chaque T-Plug"
       get SV(LSV);       "On arrive au C-Plug de la cellule connectée"
...... (calcul de Ct_SV) .......
     > "close read thru CX"
  > "close skip thru TV"
```

3) – Accès au contenu des blocs de données

Un certain nombre de classes de blocs de données sont prédéfinies dans ZOOM : par exemple, un bloc de données spécifiques au processeur de transfert (classe TR), au processeur de cellule (classe CL), au C–Plug (classe SV), au T–Plug (classe TV), aux connexions entre C–Plug et T–Plug (classe CX).

Les instructions (from, get, skip thru, read thru) permettant d'accéder à un bloc donné ou deux champs :

- le premier (deux lettres) est la classe auquelle appartient le bloc de données que l'on recherche
- le deuxième (entre parenthèses) est "l'adresse" du bloc de données correspondant à la classe voulue, trouvée en suivant les chaînages (définie au début du processeur) et à partir de la "racine" courante.

Cette "adresse" permet d'accéder aux variables prédéfinies dans la classe à laquelle appartient ce bloc de données, par l'instruction suivante :

`(LBank).Var_Name` avec : `LBank` : "adresse" du bloc de données
`Var_Name` : nom de la variable prédéfinie dans la classe à laquelle appartient ce bloc de données

Les principales variables prédéfinies, utiles pour écrire un processeur, sont les suivantes (les noms des variables contenant l'adresse du bank correspondent au nom couramment utilisé, mais peuvent être quelconques).

```
Block de données attaché a chaque cellule : CL(LCL)
(LCL).Types(2)      :  code derivant la ”Nature” du processeur
(LCL).Nb_St_Var     :  Number of state variables
(LCL).Nm_Cl         :  Cell first name
(LCL).Cd_Cl         :              and code
Bloc de données attaché a chaque transfert :TR(LTR)
(LTR).Types(2)      :  code derivant la ”Nature” du processeur
(LTR).Nb_Tr_Var     :  Number of transfer variables
(LTR).Nb_Extr       :  Number of extremities
(LTR).Nb_Fluxes     :  Number of fluxes
Block de données attaché a chaque C−Plug :  SV(LSV)
(LSV).Nb_St_Var     :  Number of State Variables connected to this C−Plug
(LSV).Cx_St_Var_Nb(1:Nb_St_Var)  :  List of connected state variable num-
bers
Block de données attaché a chaque T−Plug :  TV(LTV)
(LTV).Nb_Cx_CX      :  Number of connected CX's
(LTV).Nb_Tr_Var     :  Number of transfer Variables connected to this T−Plug
(LTV).Cx_Tr_Var_Nb(1:Nb_Tr_Var) : List of connected transfer Variables num-
bers
```

4) – Accès aux variables réservées

L'instruction **grasp array** permet d'accéder aux variables réservées (vecteurs et matrice) spécifiées dans l'instruction. Ces instructions sont exécutables et doivent donc être présentes avant d'accéder aux variables.

Pour une cellule

- Pour pouvoir accéder aux vecteurs ETA et GAMMA :

```
Grasp array ETA, GAMMA (LCL) ;
```

Cette instruction est en général placée au début de chaque point d'entrée où l'on utilise l'un de ces vecteurs.

- Instructions situées à l'intérieur de la boucle sur les $\vec{\varphi}_{TV}$ (boucle présentée précédemment) :

```
Grasp array FI_TV (LTV) ;
Grasp array ABt_TV (LCL, LTV) for (LCX) ;
```

Pour un transfert sans matrice D

- Pour pouvoir accéder aux vecteurs FI et OMEGA :

```
Grasp array FI , OMEGA (LTR) ;
```

Cette instruction est en général placée au début de chaque point d'entrée où l'un de ces vecteurs est utilisé.

- Instructions situées à l'intérieur de la boucle sur les $\vec{\eta}_{SV}$:

```
Grasp array ETA_SV (LSV) ;
Grasp array CT_SV (LSV , LTR) for (LCX) ;
```

5) – Variables et paramètres propres à chaque processeur

- Un bloc de données (bank CD pour les cellules ; TD pour les transferts) est dédié aux variables et paramètres des processeurs.
- Du point de vue de l'utilisateur, on distingue deux catégories de variables :
- Les variables directement rattachées aux processeurs ;
- Les variables "déportées", c.à.d. des variables qui existent déjà par ailleurs (dans le bloc de données d'un autre processeur, dans un vecteur $\vec{\varphi}_{TV}$ ou $\vec{\eta}_{SV}$, ...) et auquel on souhaite pouvoir accéder aisément.

- Variables directement rattachées au processeur (cf. variables "déportées", ci-dessous).
  Ces variables sont directement stockées dans le bloc de données CD ou TD. Il existe deux objets informatiques différents pour ce type de variable (`var et local_var`), strictement équivalents d'un point de vue utilisateur mais légèrement différents d'un point de vue informatique.
- Il est vivement recommandé de respecter la règle suivante : Les variables entières, utilisées pour dimensionner une (ou plusieurs) autre variable, doivent être des local–var. Elles doivent être déclarées avant d'être utilisées pour le dimensionnement de variables (en cas de nécessité, des dérogations peuvent être demandées, et même parfois être accordées).

Pour des raisons "d'efficacité", il est préférable de suivre également les règles suivantes :

- les **`local_var`** sont déclarées en premier ;
- toutes les variables non dimensionnées sont des **`local_var`**
- les variables dimensionnées sont des **`var`**

• *Variables "déportées"*

La variable elle-même est stockée à l'extérieur du bloc de données CD ou TD du processeur. Dans le bloc de données on établit un lien (un pointeur) sur cette variable. On crée une variable "fictive" (une **`link_var`**) qui utilise ce lien et permet d'accéder directement à la variable "déportée".

Exemple :

Le processeur de transfert CD1 correspnd à $Flux = h\left(T(1) - T(2)\right)$ avec :

$T(1)$: 1ère composante du $\vec{\eta}_{SV}$ connecté au 1er T-Plug

$T(2)$: 1ère composante du $\vec{\eta}_{SV}$ connecté au 2ème T-Plug

Plutôt que d'avoir, à chaque pas de temps, à parcourir les $\vec{\eta}_{SV}$ et chercher les valeurs, on peut pour simplifier utiliser des **`link_var`**.

```
! On déclare une link_var T de dimension 2
  link_var T(2) : real in class TD;
  ....
! On definit les liens entre les link_var et les variables déportées
boucle_sur_les_Eta_SV_connectés
< grasp array ETA_SV(LSV); "pour pouvoir accéder à ETA_SV"
  Si (on passe par le 1er T-Plug)
  < link T(1) to ETA_SV(1);
  > Sinon_Si (on passe par le 2eme T-Plug)
  < link T(2) to ETA_SV(1);
  >
> "fin de boucle"
....
"ensuite on peut faire directement"
Flux = H * ( T(1) - T(2) );
```

- Déclaration d'un bloc de données TD ou CD

Cas d'un processeur de cellule; pour un transfert , même chose mais avec partout TD au lieu de CD

```
!
!  === Cell parameters ===
implicit access to bank CD(LCD);"Pour pouvoir accéder aux variables
                                 déclarées dans ce bloc comme à des
                                 variables FORTRAN classiques"
declare  class CD;
local_var List_de_variable : Type_des_var in class CD;
var       List_de_variable : Type_des_var in class CD;
link_var  List_de_variable : Type_des_var in class CD;
end declare class CD;
!
```

avec ({ } voulant dire optionel) :

* `List_de_variable : Var_name { ( nbcol { ,nblin } ) } {,Var_name_2....}`
  `Var_Name :` Nom de variable. Doit etre compose uniquement de caracteres alphanumeriques ou du caractere souligne. Pas de restriction sur la longueur.
  `nbcol, nblin :` nombre de colonnes, nombre de lignes. Ces dimensions peuvent etre des expressions contenant des nombres entiers, des PARAMETER FORTRAN, une autre variable (deja definie) du meme bank; dans ce cas, cette variable doit etre entre crochet.
* `Type_des_var :` type de la variable `: INTEGER , REAL.` Si le type n'est pas defini, on prend la convention FORTRAN : entier pour les variables dont la premiere lettre est de I a N, reel sinon.

- Création du bloc de données

Cas d'un processeur de cellule.

```
! il faut tout d'abord que l'ensemble des variables utilisées dans une dimension
! soient instantiées par l'instruction SET :
  SET .Var_Name for CD = Valeur;
! création du block de données
  Lift LCD of class CD under (2,0);
! on lie ce block de données au block CL de la cellule
```

@LCL@CL@CD@ = LCD; "Link CDesc"
Pour un transfert , même chose mais avec partout TD au lieu de CD

• Liaison d'une link_var

* Pour faire "pointer" une `link_var` du `CD` ou `TD` bank du processeur où l'on est sur une variable déclarée dans un `CD` ou `TD` bank d'un autre processeur .

```
link Var1 to Var2 of (LProc);
```

avec `Var1` : nom de la `link_var` "locale"
`Var2` : nom de la `var` ou `local_var` "externe"
`LProc`: adresse permettant d'accéder à cette variable "externe"
(`LProc` = `LSV` ou `LCL` ou `LTV` ou `LTR`). Ne doit en aucun cas correspondre au processeur où l'on est (FORTRAN n'étant pas réentrant),

* Pour faire "pointer" une `link_var` du `CD` ou `TD` bank du processeur où l'on est sur une variable "réservée" (apres avoir fait un `Grasp Array` de cette variable)

```
link Var1 to Reserved_Var(n);
```

avec `Var1` : nom de la `link_var` "locale"
`Reserved_Var` : `Eta_SV` , `Fi_TV` , `Fiz_TV`
`n` : indice dans la `Reserved_Var`

* Pour faire "pointer" une `link_var` du `CD` ou `TD` bank du processeur ou l'on est sur une variable de ce même processeur.

```
link Var1 to Var2 ;
```

avec `Var1` : nom de la `link_var` "locale"
`Var2` : nom de la var ou `local_var` "locale"

• Lecture d'une variable d'un autre objet

Pour lire une variable déclarée dans un CD ou TD bank d'un autre processeur et la ranger dans une variable Fortran
set X = **[**Var2**]{:**type} **of** (LProc); " {...} veut dire optionel "
avec `X` : nom de la variable Fortran (ou d'une variable déclarée dans le CD ou TD bank local, si on a fait un `IMPLICIT ACCESS` ...)
`Var2` : nom de la `var` , `local_var` ou `link_var` "externe"
`type` : I ou R ; type de la `var`, `local_var` ou `link_var` "externe". Si le type n'est pas spécifié, on prend le type FORTRAN par défaut (entier de 1 à N, réel sinon) de la variable externe `Var2`. Diagnostique d'erreur si le type (precise ou par defaut) ne correspond pas au type de `Var2` tel qu'il a été précisé lors de sa déclaration.
`LProc` : Adresse permettant d'accéder a cette variable "externe"
(LProc = LSV ou LCL ou LTV ou LTR)

* ACCES au block de données
!!!!!! avant d'acceder au bloc de données CD ou TD, ne pas oublier de faire un get CD(LCD) ou un get TD(LTD) (en général on le fait systématiquement au début de chaque point d'entrée). Results may be hazardous sinon. !!!!!!!!!

Exemple :

On reprend plus completement l'exemple precedent de du transfert CD1

```
SUBROUTINE TCD1(LTR);
......
! *** définition du block de données
implicit access to bank TD(LTD);
declare class TD;
```

```
var
    Conduct : real in class TD;  " conductance"
link_var
    T_Edge(2) : real in class TD; "Connected State"
end declare class TD;
!
from TR(LTR)
< .........
! *** création du block de données
  Lift LTD of class TD under (2,0);
! *** on lie ce block de données au block TR du transfert
  @LTR@TR@TD@ = LTD;  "Link TDesc"
.....................
! *** on fait pointer les link_var T_Edge(1) et T_Edge(2) sur les deux variables
! *** d'etat ad hoc
  skip thru TV(LTV)       "loop on T-Plugs"
  < NCX = (LTV).Nb_Cx_CX;
    read thru CX(LCX) (J=1,NCX)   "loop on connexions"
    < NCXT = NCXT+1; get SV(LSV);
      grasp array Eta_SV (LSV);
      link T_Edge(NCXT) to Eta_SV(1); "first variable should be the temperature"
    > "close read thru CX"
  > "close skip thru TV"
  if (NCXT.NE.2)
  < PRINT *,'*** ERROR: TR SHOULD HAVE TWO CONNECTIONS, not ',NCXT ;
  >
.........................
! *** utilisation des link_var
 FI(1) = Conduct*(T_Edge(1)-T_Edge(2));
> "close from TR"*
```

VI – LES DIFFÉRENTS POINTS D'ENTRÉE D'UN PROCESSEUR

Pour la chronologie précise d'appel de ces points déntrée, voir § VII.

1) – Zinit entry

- Egalement appelé pour les cellules "I entry" et pour les transferts "Y entry".
- Ce point d'entrée est appelé par ZINIT lors de la création d'un transfert ou d'une cellule.
  `create TR, type = tr-type` ; génère entre autre un call `Y tr-type`
  `create CL-Name, type = cl-type` ; génère entre autre un `call I cl-type` (voir ZDL Manual)
- Le but de ce point d'entrée est la création d'une part d'un bloc de données (bank CL ou TR) contenant toute l'information relative à une cellule ou un transfert (nom, nombre de variaibles d'état ou de transfert, code du processeur, etc.) et, d'autre part, d'un bloc de données dans lequel sera rangé le vecteur d'état ou de transfert (bank ETA ou FI).
- Pour un processeur de transfert, on a typiquement :

```
Zinit_Entry GENE1
             ( Types(2)    = 2 04 01 01 00,
                Nb_Tr_Var   = 1,
                Nb_Extr     = 1,
                Nb_Fluxes   = 1 );
Lift LTD of class TD under (2,0);
@LTR@TR@TD@=LTD;   "Link TDesc"
! Si on a des paramateres en sortie graphique, decommenté la ligne suivante
!  Lift LPW of class PW under (LTD,OTDPW);
RETURN;
```

- Pour un processeur de cellule, on a typiquement :

```
Zinit_Entry HAIR
         ( Types(2)   = 2 04 01 01 18,
           Nb_St_Var  = 4  );
Set .NbMaxEdge  for CD = 10 ; "**** a modifier eventuellement *****"
Lift LCD of class CD under (2,0);
@LCL@CL@CD@=LCD;    "Link CDesc"
! Si on a des paramateres en sortie graphique, decommenté la ligne suivante
!  Lift LPW of class PW under (LCD,OCDPW);
return;
```

- Le code spécifiant la "nature" du processeur répond à une syntaxe extrêmement précise. Heureusement le non respect de cette syntaxe est aussi courant que le code est inutilisable.

```
************************************
*** NUMEROTATION DES PROCESSEURS ***
************************************

Code a 9 chiffres, compose de 5 champs :

Code :     |.|..|..|..|..|
Champs      0  1  2  3  4

champs   0 : code labo (1 chiffre)
         1 : code programmeur (2 chiffres)
```

```
         2 : identification externe (ultime?) du processeur (2 chiffres)
         3 : mode de representation des variables (2 chiffres)
         4 : nature des variables (2 chiffres)

=======================================================
===    codes du champ 3 :  REPRESENTAION DES VARIABLES    ===
=======================================================

Pour les cellules :
-------------------
 01 : Noeud
 02 : positions de source
 03 : Valeur moyenne/position

Pour les transferts :
---------------------
on rajoute 90 lorsqu'il y a reduction de chi2
 01 : Flux
 02 : Flux cumules
 03 : Variable d'etat d'interface
 04 : Flux et var. d'etat d'interface
 05 : Flux cumules et var. d'etat d'interface
 99 : K-transfert

=======================================================
===        codes du champ 4 :  NATURE DES VARIABLES       ===
=======================================================

 00 : quelconque
 01 : masse
 02 : energie
 03 : impulsion
 04 : temperature
 05 : vitesse
 06 : vorticite
 07 : masse/energie/impulsion
 08 : masse/energie/vorticite/impulsion
 09 : energie/impulsion
 10 : position
 11 : vitesse/position
 12 : pression position
 13 : pression
 14 : vitesse/temperature
 15 : pression/temperature
 16 : force
 17 : temperature/pression eau
 18 : Flux chaleur / Flux humidite
 19 : Flux radiatif IR
 20 : Flux radiatif visible
 21 : temperature/concentration
 22 : Flux humidité
 23 : Longueur(épaisseur,hauteur...)
 60 : fichier meteo
```

2) – Initialization of structure parameters entry point

- Egalement appelé pour les cellules “R entry” et pour les transferts “S entry”
- Ce point d’entrée est appelé après OFRAME (voir § VII), donc après que l’ensemble de la structure définitive de l’arbre ait été mis en place.

- Les opérations classiquement effectuées dans ce point d'entrée sont les suivantes :
- On vérifie que les connexions effectuées respectent les exigences du processeur
- On fait pointer les link-var sur les bonnes variables
- On calcule les paramètres constants, le cas échéant
- Si un processeur de transfert a une matrice D, faire un grasp away D-TV, même si l'on ne remplit pas la matrice D : ça positionne quelques flag utilisés ensuite par ZOOM.
- Les processeurs de transfert doivent calculer FI (utile si le transfert est gelé ou lors de la résolution de l'équation de transfert [lorsqu'un transfert a une matrice D]).
- *Exemple* : point d'entrée "S" du processeur de transfert CD1.

```
ENTRY SCD1(LTR);
  Print_TR_ID(LTR) ;  "Print transfert Id"
  get TD(LTD);        "Annable access to TD bank"
  grasp array FI,OMEGA (LTR);  "Annable access to vector FI"
  call PrTrCx(LTR);   "Print transfert connexions"
  NCXT = 0 ;
  skip thru TV(LTV)   "Loop on T_Plugs"
  < NCX = (LTV).Nb_Cx_CX;  "number of T-Plug connexions"
    if (NCX.ne.1)
    < PRINT *
         ,'*** ERROR : only one connection per T-Plug allowed, not ',NCX ;
    >
    read thru CX(LCX) (J=1,NCX)   "Loop on T-Plug connexions"
    < NCXT = NCXT+1; get SV(LSV);   grasp array Eta_SV (LSV);
      link T_Edge(NCXT) to Eta_SV(1);
    > "close read thru CX"
  > "close skip thru TV"
  if (NCXT.ne.2)
  < PRINT *,'*** ERROR: TCD1 SHOULD HAVE TWO CONNECTIONS, not ',NCXT ;
  >
  FI(1) = Conduct*(T_Edge(1)-T_Edge(2));
  Z2_Pr : Conduct, PrtLevel,FI(1);      "Print the transfert parameters"
RETURN;
```

3) – Simulation Entry

- Egalement appelé "C entry" pour les cellules et "T entry" pour les transferts
- Ces points d'entrée sont appelés à chaque pas de temps pour tous les objets qui n'ont pas été "gelés" dans ZINIT.

a) – Processeur de transfert (sans matrice D)

- L'état de toutes les cellules connectées correspond à l'état au début du pas de temps
- Le processeur doit calculer :

- les variables de transfert (vecteur FI)
- la matrice C_{SV}^{T} $(CT - SV)$ associée à chacun des vecteurs $\vec{\eta}_{SV}$ $(ETA - SV)$ connectés à ce transfert
- le vecteur $\delta\ \vec{\varphi}_{ins}$ $(OMEGA)$

b) – Processeur de cellule

- Les variables de tous les transferts connectés correspondent aux valeurs au début du pas de temps
- Le processeur doit calculer :

- la matrice $A^{-1}\ B^{T}_{TV}$ $(ABt\ -\ TV)$ associée à chacun des vecteurs $\vec{\varphi}_{TV}$ $(FI\ -\ TV)$ connectés à cette cellule

- le vecteur $\delta\ \vec{\eta}_{dec}$ $(GAMMA)$

4) – State dependent parameters entry (advanced programmer only)

Egalement appelé “K entry” pour les cellules. Le frère jumeau pour les transferts (“F entry”) est actuellement inutilisé.

Point d’entrée, généralement vide. Il est préférable, pour y effectuer des opérations, que l’utilisateur ait une bonne connaissance de la chronologie des calculs dans ZOOM.

Ce point d’entrée a deux objectifs :

- calculer les paramètres de la cellule, dépendants de l’état de la cellule et utilisés par les processeurs de transfert ;
- permettre d’assurer une dépendance explicite entre variables d’état (la résolution par la navette n’assurant la dépendance entre variables d’état que si l’hypothèse de linéarité est parfaitement vérifiée).

5) – Parameter Entry

Juste avant l’instruction END, tous les processeurs doivent avoir l’instruction Parameter-Entry; , qui génère des instructions FORTRAN permettant d’accéder directement aux variables des blocs CD et TD depuis ZINIT ou depuis un autre processeur.

6) – Exemple complet de processeurs

```
!==========================================================================
  SUBROUTINE TCD1(LTR);
!==========================================================================
  .....
LOCAL_VALID_CHAINS: [~TR~TV~CX(.)~SV~ET~],[~TR~TD],
                    [~CX~IC~CT~],[~TR~FI~],[~TR~MG~],
END_CHAINS;
!
***************************************************************************
! *                        CONDUCTIVE TRANSFER .
***************************************************************************
!  + Transfer corresponding to continuity equation
!       of type Flux = Landa * Grad(Eta)
!  + Flux = Conduct * ( T_Edge(1) - T(Edge(2) )
!  + The transfer sign convention is positive from first to second TPlug
!  + Non cumulated transfer
!  + ZINIT-inicialisacao :    parametro CONDUCT
!
==========================================================================
! ===  parameters ===
implicit access to bank TD(LTD);
declare class TD;
var
    Conduct : real in class TD;  " conductance"
link_var
    T_Edge(2) : real in class TD; "Connected State"
end declare class TD;
!
!==== Simulation Entry Point
!===========================================================
from TR(LTR)
< get TD(LTD);  "Enable access to TD bank"
  if Zprint  < Print_TR_ID(LTR) >;
  grasp array FI,OMEGA (LTR);  "Enable access to FI and Omega vectors"
  TSIGN = 1.;
  skip thru TV(LTV)  "Loop on Tplug"
  < NCX=(LTV).Nb_Cx_CX;
    read thru CX(LCX) (J=1,NCX)  " Loop on T-Plug's connexions"
    < get SV(LSV);      "C-Plug is there"
      grasp array Ct_SV (LSV,LTR) for (LCX);   "Enable access to Ct_SV"
      Ct_SV(1,1) = Conduct * TSIGN ;
      TSIGN = -TSIGN;
    > "close read thru CX(LCX)"
  > "close skip thru TV(LTV)"
  FI(1) = Conduct * (T_Edge(1)-T_Edge(2));
  Omega(1) = 0. ;
  if Zprint < Z2_Pr : FI(1) , T_Edge(1) , T_Edge(2); >
RETURN;
!=====================================================
!   Entry for transfer dependant parameters computation
ENTRY FCD1(LTR);
RETURN;
!
! ==== Initialization Entry Point, called by Zinit
!==============================================================
Zinit_Entry CD1 (
!===============
    Types(2)    = 01 01 02,
    Nb_Tr_Var   = 1,
```

```
    Nb_Extr      = 2,
    Nb_Fluxes    = 1 )
!
  Lift LTD of class TD under (2,0);
  @LTR@TR@TD@ = LTD;  "Link TDesc"
RETURN;
" "
! ==== Initialization of structure parameters Entry Point,
!        called after all tree-connections done
!==============================================================
ENTRY SCD1(LTR);
  Print_TR_ID(LTR) ;      GET TD(LTD);
  grasp array FI,OMEGA (LTR);
  call PrTrCx(LTR);  "Print transfert connexions"
  NCXT=0; " Nb of Cx:=Connections"
  skip thru TV(LTV)
  < NCX=(LTV).Nb_Cx_CX;
!   This transfer has no cumulated flux and must have 2 T-Plugs
!   (The transfer sign convention is positive from first to second TPlug).
    if (NCX.ne.1)
    < PRINT *
         ,'*** ERROR : only one connection per T-Plug allowed, not ',NCX ;
    >
    read thru CX(LCX) (J=1,NCX)
    < NCXT = NCXT+1; GET SV(LSV); GRASP ARRAY Eta_SV (LSV);
      link T_Edge(NCXT) to Eta_SV(1);
    >
  >
  if (NCXT.NE.2)
  < PRINT *,'*** ERROR: TR SHOULD HAVE TWO CONNECTIONS, not ',NCXT ;
  >
  FI(1) = Conduct*(T_Edge(1)-T_Edge(2));
  Z2_Pr : Conduct, FI(1);
RETURN;
>;
Parameter_Entry;
END;
```

```
!==============================================================
   SUBROUTINE TGENE1(LTR);
!==============================================================
   .......
LOCAL_VALID_CHAINS: [~TR~TV~CX(.)~SV~ET~],[~TR~TD],
                    [~CX~IC~CT~],[~TR~FI~],[~TR~MG~],
END_CHAINS;
!*********************************************************************
!       generateur de courant  I = Val_Cst + A * sin ( Om * t )
!*********************************************************************
! Flux non cumulé
! Valeur a initialiser dans ZINIT :  Val_Cst   , A   , Om
!*********************************************************************
!  ===  parameters ===
implicit acces to bank TD(LTD);
declare class TD;
var Val_Cst            "Fixed value"
  , A                  "Amplitude"
  , Om                 "Pulsation Om because Omega reservé"
     :real in class TD;
end declare class TD;
!
!==== Simulation Entry Point
!===============================================================
FROM TR(LTR)
<  IF ZPRINT  < Print_TR_ID(LTR) >;
  GRASP ARRAY FI,OMEGA (LTR);    GET TD(LTD);
  "Ct matrix is zero"
  FI(1) = Val_Cst + A * sin( Om * time);
  Omega(1) = A * Om * cos( Om * time) * DTIME;
  if ZPRINT < Z2_Pr : FI(1), Omega(1); >
RETURN;
!================================================================
!   Entry for transfer dependant parameters computation
ENTRY FGENE1(LTR);
RETURN;
!
!==== Initialization Entry Point, called by Zinit
!===============================================================
Zinit_Entry GENE1
             ( Types(2)    = 2 04 01 01 00,
                Nb_Tr_Var   = 1,
                Nb_Extr     = 1,
                Nb_Fluxes   = 1 );
Lift LTD of class TD under (2,0);
"Link TDesc" @LTR@TR@TD@=LTD;
RETURN;
!
!==== Initialization of strucure parameters Entry Point, called after Zinit
!==========================================================================
ENTRY SGENE1(LTR);
  Print_TR_ID(LTR) ;    GET TD(LTD);
  call PrTrCx(LTR);     GRASP ARRAY FI,OMEGA (LTR);
  FI(1) = Val_Cst + A * sin( Om * time);
  Z2_Pr : Val_Cst , A , Om , FI(1), Omega(1);
RETURN;
>;
Parameter_Entry;
END;
```

```
    SUBROUTINE CND01(LCL);
!================================================================
!*     Capacitance Processor same as CND1 , but without      *
!* steady state solution                                     *
!================================================================
! State equation : Capa * dT/dt - Flux_Somme = 0
! + One state Variable
! + The first component of each Tplug is a Flux
! + The transfer sign convention is positive from first to second TPlug,
!   if the transfer has 2 extremity
! + Transferts with more then 2 extremity must give a positive inwards flux
! + Parametre a initialiser dans ZINIT : Heat_capa
!================================================================
    .......
LOCAL_VALID_CHAINS:[~CL~CD~],[~CL~ET~],[~CL~GA~],[~CL~AB~],
                   [~CL~SV~CX~TV~TR~], [~SV~CX~IC~],
                   [~TR~TV~CX~],       [~CX~TV~FI~],
END_CHAINS;
!  === Cell parameters ===
implicit access to bank CD(LCD);
declare class CD;
var
    Heat_capa : real in class CD;
end declare class CD;
!==== Simulation Entry Point
!===============================================================
from CL(LCL)
< get CD(LCD);    "Enable access to CD bank"
  grasp array Eta,Gamma (LCL);  "Enable access to Eta and Gamma vectors"
  if ZPrint  < Print_CL_ID(LCL) >;
  Dtime_2 = .5 * DTIME;     Dtime_1 = DTIME;  "Flux non cumules"
  SumFiT=0.;      " Sum of Heat Flux "
  Skip thru SV(LSV)  "Loop on C-Plugs"
  < skip thru CX(LCX)  "Loop on C-Plugs's connexions"
    < from TV(LTV)       "T-Plug is there"
      < if ((LTV).Nb_Tr_Var.EQ.0)
                         TO NEXT LCX; "if dim(FI_TV)= 0 next FI_TV"
        grasp array FI_TV(LTV);       TSIGN = Sign_TV(LTV);
        grasp array ABt_TV (LCL,LTV) for (LCX);
        SumFiT = SumFiT + TSIGN * FI_TV(1);
        ABt_TV(1,1) = -TSIGN * Dtime_2 / Heat_capa ;
      > "close from TV(LTV)"
    > "close skip thru CX(LCX)"
  > "close skip thru SV(LSV)"
  if Zprint < Z2_Pr : SumFiT ; >
  Gamma(1) = SumFiT * Dtime_1 / Heat_Capa ;
RETURN;
!==================================================
!   Entry for state dependant parameters computation
ENTRY KND01(LCL);
RETURN;
!
!==== Initialization Entry Point, called by Zinit
!===============================================================
Zinit_Entry ND01
               ( Types(2)   = 01 01 04,
                 Nb_St_Var  = 1  );
  set_signed_ABt_TV(LCL);
  Lift LCD of class CD under (2,0);
  @LCL@CL@CD@=LCD;  "Link CDesc"
RETURN;
!==== Initialization of strucure parameters Entry Point, called after Zinit
!==========================================================================
ENTRY RND01(LCL);
  GET CD(LCD);     Print_CL_ID(LCL);
  Z2_Pr : Heat_capa, PrtLevel;
RETURN;
>; "close from CL(LCL)"
Parameter_Entry;
END;
```

VII – CHRONOLOGIE DES PRINCIPALES ÉTAPES DE CALCUL ET D'IMPRESSION DANS ZOOM

Phase d'initialisation

Principale routine : ZINIT, OFRAME, début de ZOOM

ZINIT (écrit pour l'utilisateur)

- Définition de la structure, de l'arbre, des objets donnés par l'utilisateur
- Appel du point d'entrée "Zinit-Entry" (ou "I" et "Y" entry) des processeurs pour chaque objet créé.
- Création des blocs de données liés au processeur : principalement CL, CD, ETA, GAMMA pour les cellules et TR, TD, FI, OMEGA pour les transferts.
- Définition par l'utilisateur de la valeur des paramètres de chacun des objets, et de l'état initial des cellules.

OFRAME

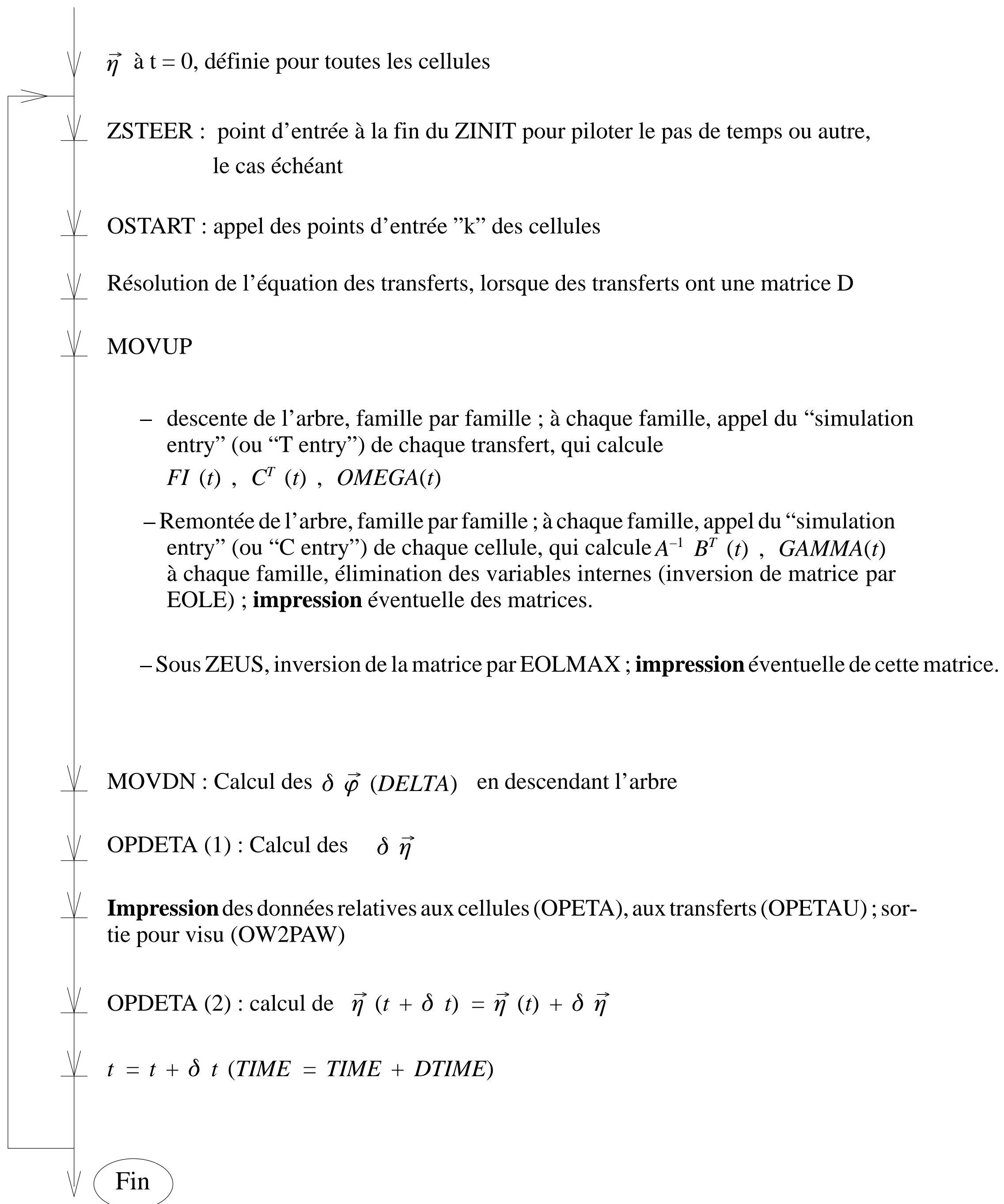
- Mise en place définitive de l'ensemble de la structure : position des transferts dans l'arbre des familles, numérotation unique des cellules, ordonnance des plug, mise en place de la structure d'accueil des matrices (face, interface, ...).

ZOOM

- Appel du point d'entrée "Initialization of structure parameter entry" ("R" et "S" entry) des processeurs pour chaque objet créé.
- Le cas échéant, initialisation de l'état des cellules, des transferts et du temps t à partir d'un fichier (instruction Resume from dans ZINIT).

⇒ Puis simulation au cours du temps.

SIMULATION AU FIL DU TEMPS



VII ALGORITHME DE LA NAVETTE *ZOOM/KER* RELATIF AUX MATRICES D DES TRANSFERTS

La navette standard – en l'absence de matrices D – résout un simple système algébrique linéarisé; elle ne nécessite donc aucune itération en régime dynamique. Par contre, la présence de matrices D couplant certains transferts entre eux par des fonctions non linéaires représente un système stationnaire à résoudre.

De manière formelle, on doit résoudre le système

$$\vec{\varphi} = \vec{f}(\vec{\varphi})$$

En effet l'état des cellules constitue pour ce problème des conditions limites figées à leur valeur à la fin du pas de temps précédent. On résout en fait :

$$\vec{\varphi} = \vec{f}(\vec{\varphi}_z) \qquad \text{(III–2.5)}$$

où on a noté $\vec{\varphi}_z$ la valeur de l'inconnue à l'itération précédente. La résolution est basée sur la linéarisation de $\vec{f}(.)$ pour calculer $\vec{\varphi}$ en fonction de $\vec{\varphi}_z$ jusqu'à convergence – méthode de Gauss; le critère de convergence est un écart quadratique, soit standard, soit héritant d'accroissements calculés dans chaque modèle partiel en fonction de critères physiques localement motivés

On différencie l'expression précédente (III–2.5) :

$$\vec{\varphi}_z + d\vec{\varphi} = \vec{f}(\vec{\varphi}_z) + \frac{\overline{\overline{df}}}{d\vec{\varphi}} d\vec{\varphi}$$

$$\left(\P - \frac{\overline{\overline{df}}}{d\vec{\varphi}} \right) d\vec{\varphi} = \vec{f}(\vec{\varphi}_z) - \vec{\varphi}_z$$

et on a donc un système linéaire à résoudre en $d\vec{\varphi}$.

Cette présentation ne rentre pas dans les difficultés liées à l'emboîtement des transferts, qui est semblable à la navette standard et comporte les mêmes étapes – et donc les mêmes possibilités de parallèlisation.

Comme pour tous les algorithmes itératifs, l'ordre des calculs est crucial pour la rapidité de convergence. Dans le cas du *TEF*, cet ordre est – a priori – induit par la structure de partition des modèles, et l'on se trouve en face d'une concurrence entre l'efficacité numérique et l'analyse des couplages telle que nous l'avons concrétisée. Nous avons cependant abouti à un compromis qui ne brise pas la modularité et qui nous permis d'accélérer l'algorithme par un facteur 2 à 3. Ceci, combiné à une initialisation plus fine des inconnues concernées par l'algorithme, a finalement permis de gagner un facteur 3 à 4 suivant les problèmes traités.

Définition de classes de Transferts

Pour induire un ordre des calculs sans remettre en cause la structure de partition des transferts, on a définit trois classes de transferts :

les transferts ne possédant pas de matrice D (Tr) mais dont dépendent des transferts possédant une matrice D (D_Tr) : classe 1 ;

les D_Tr dépendants d'autres transferts mais sans réciprocité : ils dépendent donc uniquement de la classe 1 : transferts de classe 2 ;

tous les autres D_Tr : classe 3 .

Le coup de parcourt d'arbre étant quasi nul, on effectue par itération 6 navettes, qui effectuent, dans l'ordre :

l'initialisation de $\vec{\varphi}_z$ de la classe 1 à partir de la valeur de $\vec{\varphi}$ du pas précedent ;

initialisation des éléments de la classe 2 à partir de la classe 1 ;

appel des éléments de la classe 2 ;

initialisation des éléments de la classe 3 à partir des classes 1 et 2 ;

appel des éléments de la classe 3 .

L'écart quadratique est alors complet et le critère est appliqué. En cas d'itération, il est inutile de rappeler les éléments de la classe 1 dont les variables ne dépendent que des états des cellules.

Les modèles de transferts sont marqués pour l'appartenance à une classe, pour des raisons de mise en pratique rapide, mais il est prévu d'effectuer ce calcul au cours d'un pré–traitement, puisque toute l'information nécessaire est disponible dans la base de donnée construite sur la structure de l'arbre de séparation.

Cette gestion simple des initialisation et calculs regroupés par classe ne permet pas d'ordonner les calculs dans chaque classe, qui hérite strictement de l'arborescence ; on peut d'ailleurs remarquer que le contraire s'opposerait à la parallélisation de l'algorithme. On a gagné en pratique un facteur 5 à 6 sur le temps calcul de cette partie lorsque le système est en régime transitoire ; lors de l'approche d'une solution stationnaire, une seule itération suffit en général pour vérifier le critère de convergence.