Usage de JEVEUX

Résumé :

Il s'agit ici d'indiquer quelques notions de fonctionnement du gestionnaire de mémoire JEVEUX afin de préciser l'usage des routines "utilisateur", d'indiquer les routines les mieux appropriées à certaines actions et de signaler les difficultés d'usage. On présente ici les règles d'usage en italique dans chaque paragraphe.

Table des matières

1 Usage des bases

Les objets simples peuvent être créés par les routines JECREO et WKVECT, les collections par la routine JECREC.

WKVECT permet d'enchaîner les trois appels JECREO, JEECRA et JEVEUO pour un objet de genre vecteur.

La notion de base permet d'associer les différents objets à un fichier sauvegardé ou non en fin de travail. Les objets à conserver en fin de travail seront créés sur la base GLOBALE associée à la classe G. Cette base permet de conserver les structures de données et d'effectuer des poursuites.

Les objets de travail seront créés sur la base VOLATILE associée à la classe V. Cette base est détruite à la fin du travail (elle est même nettoyée entre chaque commande). Par convention, on utilisera les caractères && au début du nom de tout objet associé à cette base.

On rappelle qu'il n'est pas possible de disposer d'objets de nom identique sur des bases différentes. Les routine JE.... ne possèdent pas parmi leurs arguments le nom de la classe et la recherche du nom s'effectue dans l'ensemble des répertoires associés aux différentes bases ouvertes.

.

base GLOBALE	nom de classe G	objets conservés
base VOLATILE	nom de classe V	objets temporaires

Remarque :

Il est aussi possible de créer dynamiquement des vecteurs d'entiers, réels, complexes, chaînes de caractères, … sans passer par jeveux. Il faut alors utiliser les « macros » AS_ALLOCATE et AS_DEALLOCATE.

Les différences entre un vecteur Fortran et un objet jeveux sont :

- Un objet jeveux peut être déchargé sur disque (puis rechargé en mémoire). Un vecteur Fortran n'existe qu'en mémoire.
- Un objet jeveux a un nom que l'on peut transmettre sous forme de chaîne de caractères.
- Il est plus rapide de créer un vecteur Fortran qu'un objet jeveux.
- Un objet jeveux peut faire partie d'une structure de données nommée (sd_maillage, sd_modele, …).

2 Accès par nom

L'accès aux objets gérés par JEVEUX est effectué à l'aide du nom. On utilise une fonction de codage qui fournit à partir du nom et de différents paramètres une clef d'accès (un entier), cette clef permet ensuite d'accéder aux différents attributs. L'accès par nom est relativement coûteux (décodage de caractères, gestion de collision, etc...) aussi conserve-t-on dans une variable le dernier nom (d'objet simple, de collection et d'objet de collection) et l'identificateur (obtenu à partir de la clef) associé, pour éviter un nouvel appel à la fonction de codage.

Remarque :

> *Il est donc recommandé d'effectuer toutes les requêtes pour un même objet* JEVEUX *de façon séquentielle afin de bénéficier de cette possibilité.*

3 Accès aux segments de valeurs

On accède au contenu d'un objet JEVEUX grâce à la routine jeveuo (ou wkvect).

Il y a deux types d'accès différents pour chacune de ces routines :
- un accès par « pointeur »
- un accès par « adresse dans une variable de référence »

Le premier accès (permis grâce à Fortran 2003) est désormais privilégié

3.1 Accès par pointeur

Lorsque l'on veut accéder au contenu d'un objet JEVEUX, on déclare une variable locale de type « pointeur » sur un vecteur du bon type.
L'appel à jeveuo associe le pointeur à la zone mémoire occupée par l'objet (celui-ci est alors ramené en mémoire si nécessaire).
On peut alors manipuler l'objet comme un simple vecteur fortran.

Exemple : Accès en lecture aux coordonnées des nœuds du maillage :

```
! déclaration de la variable locale :
real(kind=8), pointer :: coordo(:) => null()

...
call jeveuo(ma//'.COORDO    .VALE', 'L', vr=coordo)
n1=size(coordo) ! Accès à la longueur de l'objet (=3*nbno)
do ino=1,nbno
  x=coordo(3*(ino-1)+1)
  Y=coordo(3*(ino-1)+2)
  ...
enddo
```

Remarque importante :
Lors de la déclaration de la variable locale (ici : coordo), il est imposé d'initialiser le pointer à « zéro » (=> null()). Sinon, le pointeur est dans un état indéterminé.

3.2 Accès par adresse

Les routines jeveuo et wkvect peuvent renvoyer à l'utilisateur une adresse relative dans l'un des tableaux ZR, ZI, ZC, ou ZK, (on notera par la suite $Z?$ l'une de ces variables Fortran). Cette adresse est valide tant qu'il n'y a pas eu de libération.

La notion d'accès en écriture ou en lecture permet d'éviter le déchargement systématique sur disque du segment de valeurs et limite ainsi le nombre des entrées/sorties sur disque. Les objets accédés en lecture ne seront pas sauvegardés sur disque au moment de la libération. L'appel à WKVECT effectue une requête en écriture.

Un segment de valeurs a pu être accédé en écriture puis libéré, le gestionnaire le conserve alors en mémoire et diffère son déchargement sur disque lors d'une prochaine recherche de place. Un nouvel accès en lecture renvoie l'adresse du segment déchargeable, une modification du contenu peut donc avoir lieu à l'insu de l'utilisateur si ce dernier affecte le contenu du tableau $Z?$ à l'adresse indiquée.

Règle d'usage :

> *L'utilisateur, lorsqu'il effectue un accès en lecture, ne doit pas modifier le contenu du tableau $Z?$ à l'adresse fournie lors de la requête et doit éviter de le passer en argument d'un sous-programme dont il n'aurait pas l'entière maîtrise.*

L'appel à la routine jeveuo renvoie, l'adresse de l'objet JEVEUX relative à une variable $Z?$ du même type que l'objet JEVEUX (cette adresse est mesurée dans la longueur du type).

Le commun normalisé doit figurer dans toute unité de programme effectuant ce type d'appel.
Depuis la version NEW11.2.2 ce commun est inséré dans les sources par l'instruction :

#include « jeveux.h »

qui viendra automatiquement substituer les instructions suivantes à la compilation :

```
INTEGER ZI
COMMON/IVARJE/ZI(1)
INTEGER*4 ZI4
COMMON/I4VAJE/ZI4(1)
REAL*8 ZR
COMMON/RVARJE/ZR(1)
COMPLEX*16 ZC
COMMON/CVARJE/ZC(1)
LOGICAL ZL
COMMON/LVARJE/ZL(1)
CHARACTER*8  ZK8
CHARACTER*16 ZK16
CHARACTER*24 ZK24
CHARACTER*32 ZK32
CHARACTER*80 ZK80
COMMON/KVARJE/ZK8(1), ZK16(1), ZK24(1), ZK32(1), ZK80(1)
```

Remarque :

Ne pas modifier le nom des variables du commun de référence.

L'accès à un segment de valeurs est réalisé de la façon suivante : si $JTAB$ désigne l'adresse renvoyée par la routine JEVEUO pour un objet de genre vecteur et de type I, $KTAB$ celle pour un objet de type C (complexe) :

.

$ZI(JTAB)$	est la première valeur d'un vecteur d'entiers,
$ZI(JTAB+I-1)$	est la I éme valeur d'un vecteur d'entiers,
$ZC(KTAB+I-1)$	est la I éme valeur d'un vecteur de complexes.

Les variables (JTAB) susceptibles de contenir une adresse JEVEUX et utilisées en argument des routines sous la forme ZI(JTAB), ZR(JTAB), etc, doivent toujours être initialisées à la valeur 1. Ainsi, si cette adresse n'est pas le résultat d'un appel à JEVEUO ou WKVECT, on pointera sur une valeur valide pour le premier élément au sens de l'accès à la mémoire. ZI(1), ZR(1) et ZC(1) sont initialisées avec une valeur pouvant provoquer une erreur ou une opération conduisant à NaN.

4 Initialisation des valeurs

Lors de l'appel à jeveuo ou à wkvect le gestionnaire de mémoire effectue une recherche de place en mémoire centrale. Si l'objet ne possède pas d'image sur disque (c'est-à-dire lors de son premier accès en écriture), le segment de valeur est initialisé suivant le type de l'objet : $0.$ pour les réels, $(0.,0.)$ pour les complexes, 0 pour les entiers, $' '$ (blanc) pour les caractères.

Règle d'usage :

Il est inutile d'effectuer une boucle d'initialisation avant la première utilisation du segment de valeurs.

5 Libération des segments de valeurs et notion de marque

Si rien n'est fait (pas d'appel à JELIBE), les segments de valeurs ramenés en mémoire y restent et cela peut conduire rapidement à sa saturation. D'un autre côté, si on libère un objet sans prendre de précautions (une marque), on risque de rendre invalide l'adresse d'un objet demandé en amont de la programmation.

Ce qui est préconisé :

- on utilise les marques pour libérer les objets (routines JEMARQ / JEDEMA).
- on libère explicitement très peu d'objets (les plus gros) ce qui permet de s'assurer que les libérations ne sont pas dangereuses (bonne connaissance de l'usage de ces objets) ;

6 Détection des écrasements mémoire

JEVEUX alloue dynamiquement chaque zone mémoire associée aux segments de valeurs lors de la première requête. Les différents segments de valeurs associés aux objets sont encadrés par 4 entiers devant et 4 derrière. Ces entiers permettent de stocker l'état et le statut, l'identificateur et la classe ainsi qu'un décalage pour gérer les types de longueur supérieure à l'unité d'adressage.

L'écrasement des entiers situés autour du segment provoque la perte de l'identité de l'objet associé au segment de valeurs. Un tel écrasement est généralement détecté à l'occasion de l'écriture sur disque du contenu de l'objet et non au moment du débordement. Il se traduit par un des messages d'erreur suivants :

```
<S> <JJLIRS> <ECRASEMENT AMONT POSSIBLE ADRESSE> nnnn
```

Dans ce cas on a écrasé un des entiers situés devant le segment de valeurs.

```
<S> <JJLIRS> <ECRASEMENT AVAL POSSIBLE ADRESSE> nnnn
```

Dans ce cas on a écrasé un des entiers situés derrière le segment de valeurs.

Règle d'usage :

Le développeur dispose alors de la routine JXVERI *pour instrumenter son code.*

Cette routine vérifie l'intégrité des valeurs de part et d'autre du segment de valeurs et peut signaler le point de rupture. Elle détecte en plus une incursion hors de la zone mémoire licite. Il est possible de mettre en œuvre à moindre frais dans la commande DEBUT ou POURSUITE l'appel à ce sous-programme avant chaque commande, ce qui permet de déterminer quelle commande effectue l'écrasement. Le développeur pourra alors, en instrumentant les routines associées à la commande, procéder par dichotomie pour déterminer la routine ou les instructions erronées.

L'écrasement peut aussi être moins important et n'affecter qu'un mot devant ou derrière le segment de valeurs, c'est le cas lorsque l'on fait une erreur d'indice dans le tableau $Z?$. Le contenu de l'usage ou du statut du segment de valeur est alors affecté, cette information peut être obtenue en consultant le résultat des impressions de la répartition en mémoire par la routine JEIMPM.

7 Agrandir un vecteur

Règle d'usage :

L'utilisateur dispose de la routine juveca *pour agrandir un objet simple de genre vecteur.*

C'est une surcouche écrite à partir des routines utilisateur JEVEUX. Elle construit un objet temporaire et détruit l'original après recopie.
Après avoir agrandi un objet, il est nécessaire de redemander un pointeur sur cet objet (jeveuo).
Ces diverses opérations peuvent être assez coûteuses, il est donc préférable de minimiser le nombre d'appels à juveca pour un même objet.
Une stratégie courante est de doubler la taille d'objet quand on constate que sa taille est insuffisante (plutôt que de l'agrandir au fur et à mesure.

8 La recopie des objets

Il est possible de recopier les objets JEVEUX sur une même base ou d'une base à l'autre. La recopie des objets simples ne pose pas de problème particulier, par contre il est plus délicat de manipuler des collections. Une collection peut s'appuyer sur un répertoire de noms externe ou un pointeur de

longueur externe. Ces objets simples doivent être créés et en partie gérés indépendamment (par exemple leur destruction doit être explicite). Leur nom peut donc être sans rapport avec le nom de la collection.

Deux cas sont possibles :

- la recopie s'effectue sur des bases différentes : les pointeurs externes seront dupliqués et deviendront des pointeurs internes à la collection,
- la recopie s'effectue sur une même base : les pointeurs externes peuvent être conservés ou bien il sont dupliqués et deviennent internes.

Si le réceptacle est déjà existant, il est détruit avant recopie.

Règles d'usage :

Utiliser `jedupo`

9 Les routines travaillant sur des groupes d'objets

L'organisation des structures de données d'Aster repose en grande partie sur les noms des objets. On manipule au sein du code des "concepts" bâtis à partir d'un nom fourni par l'utilisateur comme résultat des commandes. Il a donc paru commode de pouvoir manipuler des groupe d'objets en fournissant une sous-chaîne de caractères, qui est recherchée dans les noms de tous les objets présents dans les répertoires.

Les routines `jedetc` et `jedupc` s'appliquent à des listes d'objets. Elles permettent, dans l'ordre, de libérer, de détruire et de dupliquer les objets de ces listes.

Ces routines offrent plus de souplesse au développeur pour gérer les objets (structures de données) mais elles sont moins efficaces que les routines « en dur » `detrsd` et `copisd`.

Règle d'usage :

Ne pas utiliser les routines `jedetc` et `jedupc`. Utiliser à la place `detrsd` et `copisd`.

10 Comment exploiter efficacement les collections contiguës et numérotées ?

Il arrive parfois que l'on crée des collections ayant un très grand nombre d'éléments. Par exemple, la connectivité du maillage est une collection qui peut avoir plusieurs centaines de milliers d'éléments (un élément par maille).
Lorsque l'on doit « boucler » sur les éléments de ces collections, l'usage des fonctions jeveux (`jeveuo`, `jelira`, ...) sur chaque objet de la collection peut devenir très coûteux.

Lorsque la collection est « contiguë » et numérotée, nous expliquons dans les deux paragraphes suivants, comment éviter l'usage des fonctions jeveux dans la boucle sur les éléments.

Cet usage plus efficace est directement lié à la présence d'un objet système : le « pointeur des longueurs cumulées ». Cet objet n'existe que pour les collections contiguës.

10.1 Comment accéder rapidement aux éléments d'une collection numérotée et contiguë ?

On suppose ici que l'on veut accéder dans une boucle à de nombreux éléments d'une collection contiguë numérotée (ici, tous les éléments).
La collection s'appelle COLLEC, elle a nbobj éléments.

10.1.1 Accès "classique" :

```
do iobj=1,nbobj
    ! récupération de la longueur de l'objet de collection (dimobj):
    call jelira(jexnum(COLLEC, iobj), 'LONMAX', ival=dimobj)
    ! récupération de l'adresse l'objet de collection (jcollec):
    call jeveuo(jexnum(COLLEC, iobj), 'L', jcollec)
    call xxxxx(zr(jcollec), ...)
enddo
```

10.1.2 Accès "optimisé" :

```
call jeveuo(COLLEC, 'L', jcollec1)
call jeveuo(jexatr(COLLEC, 'LONCUM'), 'L', jlc_collec)

do iobj=1,nbobj
    ! calcul de la longueur de l'objet de collection (dimobj):
    dimobj=zi(jlc_collec+iobj)-zi(jlc_collec+iobj-1)
    ! calcul de l'adresse l'objet de collection (jcollec):
    jcollec=jcollec1-1+zi(jlc_collec+iobj-1)
    call xxxxx(zr(jcollec), ...)
enddo
```

10.2 Comment créer rapidement une collection numérotée et contiguë ?

On suppose ici que l'on veut créer une collection (COLLEC) contiguë numérotée ayant nbobj éléments. Les éléments de la collection ont a priori des longueurs différentes.
On suppose également que l'on a calculé au préalable la longueur cumulée (ltot) de TOUS les éléments de la collection.

10.2.1 Création "classique" :

```
call jecrec(COLLEC, 'G V I', 'NU', 'CONTIG', 'VARIABLE', nbobj)
call jeecra(COLLEC, 'LONT',ival=ltot)
do iobj=1,nbobj
    call jecroc(jexnum(COLLEC, iobj))
    dimobj = ...
    call jeecra(jexnum(COLLEC, iobj), 'LONMAX', ival=dimobj)
    call jeveuo(jexnum(COLLEC, iobj), 'E', jcollec)
    ! remplissage de l'objet iobj :
    zi(jcollec)= ...
enddo
```

10.2.2 Création "optimisée" :

```
call jecrec(COLLEC, 'G V I', 'NU', 'CONTIG', 'VARIABLE', nbobj)
call jeecra(COLLEC, 'LONT',ival=ltot)
call jeveuo(COLLEC, 'E', jcollec1)
call jeveuo(jexatr(COLLEC, 'LONCUM'), 'E', jlc_collec)
zi(jlc_collec)=1
do iobj=1,nbobj
    dimobj = ...
    ! adresse de l'objet iobj (jcollec):
    decal=zi(jlc_collec-1+iobj)
    jcollec=jcollec1-1+decal
    ! remplissage de l'objet iobj :
    zi(jcollec)=...
    ! équivalent du jelira / LONMAX :
    zi(jlc_collec-1+iobj+1)=decal+dimobj
enddo
! Pour remplacer TOUS les "jecroc" :
call jeecra(COLLEC,'NUTIOC',nbobj)
```

Remarque importante :

L'écriture de NUTIOC est une opération un peu coûteuse. Il est donc très important de ne faire cette opération qu'une seule fois (à la fin de la construction de la collection) et non pas dans la boucle sur les éléments.