Introduire une nouvelle structure de données

Résumé :

Ce document décrit la méthode pour introduire une nouvelle structure de données dans *Code_Aster* , en particulier la rédaction au format « python » du catalogue de la structure de données.

Table des Matières

1 Introduction

Il y a deux grands type de structures de données (SD) dans *Code_Aster* :

- Les SD locales aux commandes. Ces SD ne sortent pas des opérateurs et l'utilisateur n'y a pas accès. Elles sont uniquement présentes dans l'espace Fortran.

- Les SD globales, aussi appelées « concepts » dans la documentation. Ces SD servent à communiquer les informations entre les opérateurs et entre les exécutions de *Code_Aster* . Elles doivent respecter des règles beaucoup plus strictes, être accompagnées d'outils de vérification de leur intégrité (mode SDVERI) et documentées (section D4 de la documentation de développement).

2 Les SD locales

Les SD locales sont sous la seule responsabilité du développeur, qui est maître de ses choix. Cependant, un certain nombre de recommandations sont données dans la documentation D2.05.01 (« Règles concernant la Structuration des Données »). Les SD locales sont nécessairement créées sur la base volatile JEVEUX , elles ne sortent pas d'une commande (la base volatile est nettoyée automatiquement par le superviseur d'exécution à la fin de chaque commande).

3 Les SD globales

Toutes les structures de données globales échangées dans *Code_Aster* existent dans l'espace Python. Il existe trois types de SD globales :

- Les SD globales strictement Python : elles n'existent que dans l'espace Python et n'ont pas leur pendant dans l'espace Fortran.

- Les SD globales strictement Fortran : elles existent dans l'espace Python mais sont simplement déclarées comme étant des types. Elles n'ont ni méthode, ni attribut spécifique (c'est-à-dire autres que ceux de la classe chapeau ASSD).

- Les SD globales mixtes. Elles existent dans l'espace Python, ont des méthodes et/ou des attributs spécifiques. De plus, elles existent dans l'espace Fortran.

La déclaration des SD globales est faite dans les fichiers code_aster/Cata/Legacy/DS/co_**.py.
Ces SD globales qui sont transmises d'une commande à une autre sont baptisées « concepts » : ce sont les objets Python connus du jeu de commandes. Les concepts dérivent tous, directement ou indirectement, de la classe chapeau ASSD. L'attribut de classe cata_sdj (comme « catalogue de structure de données Jeveux ») précise la classe Python qui définit la structure de données Fortran.
Par exemple :

```
cata_sdj = 'SD.sd_fonction.sd_fonction_aster'
```

qui indique que la structure de données Fortran des fonctions est définie par la classe sd_fonction_aster du module sd_fonction.py dans la bibliothèque SD.

Par la suite, on s'intéresse aux objets JEVEUX des SD et à leur vérification.

Le mécanisme de vérification des SD globales est activé dans la plupart des cas-tests, sauf lorsque cette vérification est trop coûteuse ou bien échoue. L'activation se fait de deux manières :

- Globalement, sur tout le fichier de commande, via le mot-clef SDVERI='OUI ' dans la commande DEBUT

- Localement, entre chaque commande, via le mot-clef SDVERI='OUI ' dans la commande DEBUG

La vérification est du ressort du développeur de la structure de données globales. Comme on utilise une classe Python, les vérifications peuvent être aussi poussées que nécessaires (nom et type des objets JEVEUX , cohérence des dimensions, SD spécialisées suivant les opérateurs, etc.).

3.1 SD purement en Python

Il en existe très peu. Il est recommandé de déclarer ce genre de classes, dans le fichier de commandes (ou dans un module Python importé dans le fichier de commandes). Il existe actuellement quatre classes de ce type. Voici leur déclaration dans DataStructure.py :

```
class no  (GEOM): pass
class grno(GEOM): pass
class ma  (GEOM): pass
class grma(GEOM): pass
```

Ces classes dérivent toutes de la classe mère GEOM décrite dans le fichier N_GEOM de la bibliothèque bibpyt/Noyau. Elles ne servent qu'à définir un type spécifique pour les entités géométriques pour la gestion des mots-clefs GROUP_MA/GROUP_NO dans le décodeur syntaxique du catalogue des commandes.

3.2 SD avec espace Fortran

Toutes ces classes sont importées dans l'entête du catalogue (code_aster/Cata/DataStructure.py). Exemples :

```
class cabl_precont(ASSD):
    cata_sdj = "SD.sd_cabl_precont.sd_cabl_precont"
```

cabl_precont est une classe purement Fortran, aucune méthode spécifique n'est définie.
cham_elem est une classe contenant, par exemple, une méthode EXTR_COMP qui lui est propre.

Les classes peuvent aussi hériter d'autres classes. Par exemple, la classe evol_noli hérite de la classe evol_sdaster , qui hérite elle-même de la classe resultat_sdaster.

Comme dit précédemment, l'attribut cata_sdj déclare la classe utilisée pour décrire et vérifier les objets JEVEUX de la SD fortran associée au concept (mode SDVERI). Ces classes sont toutes regroupées dans le répertoire (bibliothèque python) bibpyt/SD, sous le nom sd_****.py.

4 Classes de description des SD

La classe Python sd_* contient donc la description de la SD en termes d'objets Fortran, elle dérive de l'objet AsBase. La classe AsBase contient :

- le nom de la SD nomj
- un attribut pour classer la SD comme étant "optionnelle" ou pas : optional
- une méthode pour attribuer le nomj : setname
- une méthode pour vérifier la SD : check
- diverses méthodes pour l'impression (surcharge de *repr*)

Par héritage, on définit l'objet JEVEUX de base: class OJB(AsBase). Cette classe contient :

- la description des attributs JEVEUX (voir D6.02.01 Gestion mémoire : JEVEUX) dans des attributs protégés
- un attribut protégé sur l'existence ou pas de l'objet JEVEUX : __exists

Comme on le voit, les attributs de la classe OJB sont protégés, on y accède via des classes dérivées de OJB qui décrivent les objets existants. Il y a trois classes de base :

- OJBVect : un objet simple au sens JEVEUX
- OJBPtnom : un objet pointeur de noms au sens JEVEUX
- OJBCollec : un objet collection au sens JEVEUX

Il existe des "alias" de ces classes, pour des raisons de compatibilité et de lisibilité :

- AsObject est un autre nom de la classe OJB
- AsPn est un autre nom de la classe OJBPtnom
- AsVect est un autre nom de la classe OJBVect
- AsColl est un autre nom de la classe OJBCollec

La classe OJBVect est dérivée en objets encore plus élémentaires, ce qui permet de s'approcher de la syntaxe de la routine Fortran WKVECT :

- AsVI : l'objet est un vecteur d'entiers INTEGER*8
- AsVR : l'objet est un vecteur de réels REAL*8
- AsVC : l'objet est un vecteur de complexes COMPLEX*16
- AsVL : l'objet est un vecteur de booléens LOGICAL*8
- AsVK8 : l'objet est un vecteur de caractères CHARACTER*8
- AsVK16 : l'objet est un vecteur de caractères CHARACTER*16
- AsVK24 : l'objet est un vecteur de caractères CHARACTER*24
- AsVK32 : l'objet est un vecteur de caractères CHARACTER*32
- AsVK80 : l'objet est un vecteur de caractères CHARACTER*80

Remarque : les types utilisés dans *Code_Aster* sont définis dans le fichier asterf.config.h

Le constructeur de ces objets peut contenir tous les attributs JEVEUX habituels. Par exemple, pour une collection :

```
TAVA = AsColl(SDNom(debut=19), acces='NU', stockage='CONTIG',
              modelong='CONSTANT', type='K', ltyp=8, )
```

5 Utilisation des classes Python pour la vérification

5.1 Nommage

En premier lieu, il convient de nommer l'objet JEVEUX qui sert de base à la SD. On rappelle qu'à chaque objet JEVEUX est attribué un nom, qui est une chaîne de caractères de longueur 24 (CHARACTER*24). On rappelle également que l'utilisateur nomme ses concepts dans le fichier de commande via une chaîne de caractère de longueur 8 (CHARACTER*8). Un des principes de base pour la construction des SD globales dans *Code_Aster* est de **toujours** préfixer le nom des données JEVEUX relatifs au concept (produit par la commande) par le nom de ce dernier. Par exemple, le maillage contient un vecteur de réels avec les cordonnées des nœuds, il est créé ainsi :

```
COOVAL = MAIL(1:8)//'.COORDO    .VALE'
CALL WKVECT(COOVAL,'G V R',3*NBNO,JCOOR)
```

Ici MAIL est le nom du concept donné par l'utilisateur (MAIL = LIRE_MAILLAGE(...)). Quand on est dans la classe Python qui va être instanciée pour le concept que l'on doit vérifier, la première chose à faire est de déterminer le nom de tous les objets qui appartiendront à la SD :

```
class sd_maillage(sd_titre):
    nomj = SDNom(fin=8)
```

Ainsi nomj sera le préfixe de tous les objets JEVEUX contenus dans la SD. On le construit en prenant les 8 premiers caractères de la SD (fin=8). Ensuite, il s'agit de vérifier la présence d'objets dans la SD. Par exemple, la sd_maillage contient obligatoirement un objet DIME, qui est un vecteur d'entiers de longueur 6 :

```
class sd_maillage(sd_titre):
    nomj = SDNom(fin=8)
    toto = AsVI(SDNom(nomj='.DIME'), lonmax=6,)
```

L'objet toto est un vecteur d'entiers (AsVI) dont l'attribut LONMAX vaut 6. On aurait pu aussi écrire de manière plus compacte :

```
class sd_maillage(sd_titre):
    nomj = SDNom(fin=8)
    DIME = AsVI(lonmax=6,)
```

Cette dernière construction (DIME = AsVI(lonmax=6,)) est une facilité offerte au développeur, basée sur le fait que le nom d'un objet JEVEUX est toujours construit de la même manière. De manière implicite, quand on écrit DIME = AsVI() , on construit une instance de nom DIME de la classe AsVI dont l'objet JEVEUX a pour nom nomj(1:8)/'.DIME ' . Cette instanciation doit être privilégiée, afin de faciliter la lecture du catalogue. Il arrive parfois que les attributs d'un objet (comme son type) puissent être variables suivant l'opérateur créant cette SD. Dans ce cas, on peut utiliser la fonction membre Parmi(). Par exemple :

```
VALE = AsVect(ltyp=Parmi(4,8,16,24), type=Parmi('C', 'I', 'K', 'R'),
                docu=Parmi('', '2', '3'),)
```

Le .VALE d'un champ aux nœuds peut contenir des valeurs complexes, entières, réelles ou même des chaînes de caractères de longueur 8, 16 ou 24.

5.2 Objets facultatifs

On peut déclarer qu'un objet est facultatif dans la SD, par exemple, il peut ne pas y avoir de GROUP_NO dans la SD maillage :

```
class sd_maillage(sd_titre):
  nomj = SDNom(fin=8)
  GROUPENO = Facultatif(AsColl(acces='NO', stockage='DISPERSE',
                               modelong='VARIABLE', type='I',))
```

La fonction `Facultatif` positionne l'attribut `optionnal` à `True`. Par défaut, tous les objets sont obligatoires. Comme le mécanisme de vérification des `SD` (`SDVERI`) parcourt l'ensemble des objets `JEVEUX` attachés au concept, il est impératif que tous les objets (obligatoires ou facultatifs) aient été déclarés dans la classe de la `SD` !

5.3 Méthode d'existence « `exists` »

Il est parfois utile de savoir si une `SD` existe. Pour cela, on peut surcharger la méthode `exists` qui retourne un booléen :

```
def exists(self):
    # retourne "vrai" si la SD semble exister (et donc qu'elle peut etre
    # vérifiée)
    return self.REFE.exists
```

Pour vérifier l'existence, le plus simple est de contrôler la présence d'un objet obligatoire (ici `REFE`).

Il est important de noter que dans la classe `AsBase` (dont dérivent toutes les autres), `exists` est un **attribut** (prenant une valeur logique `True` ou `False`). Il est construit en faisant appel au plus bas niveau à la routine Fortran `JEEXIN(OBJET,IRET)`.
Lorsque `exists` est surchargé comme ci-dessus, il devient une **méthode**. Dès lors il est impératif de l'appeler comme telle, c'est-à-dire sans oublier les parenthèses ouvrantes et fermantes « () ».
Par exemple la `sd_ligrel` redéfinit la méthode `exists`, on doit donc l'appeler ainsi :

```
if self.contact_resolu():
    # ne pas oublier les () car sd_ligrel.exists est une méthode
    assert self.LIGRE.exists()
```

5.4 Méthodes de vérification « `check_` »

Toutes les classes dérivées de `AsBase` contiennent la méthode `check`. Par défaut, cette fonction se contente de vérifier la conformité de la `SD` aux attributs de l'objet `JEVEUX`. Par exemple :

```
class sd_maillage(sd_titre):
  nomj = SDNom(fin=8)
  DIME = AsVI(lonmax=6, )
```

On se contente de vérifier que l'objet `JEVEUX nomj(1:8)//'.DIME '` est bien un vecteur d'entiers de longueur 8. Il est néanmoins possible (et souhaitable !) de surcharger une méthode `check` pour faire des vérifications plus poussées. Par exemple, toujours dans la `sd_maillage`, on voudrait vérifier que le pointeur de nom `MAIL(1:8)//'.NOMNOE'`, qui contient le nom des nœuds est bien de longueur égale au nombre de nœuds :

```
class sd_maillage(sd_titre):
  nomj   = SDNom(fin=8)
  DIME   = AsVI(lonmax=6,)
  NOMNOE = AsPn(ltyp=8)

  def check_NOEUDS(self,checker):
      dime     = self.DIME.get()
```

```
        nb_no    = dime[0]
        assert self.NOMNOE.nomuti == nb_no
```

On a bien déclaré que l'objet NOMNOE était un pointeur de noms (contenus dans des chaînes de longueur ltyp=8). Puis on déclare une nouvelle méthode check_NOEUDS, dont un des arguments est obligatoirement checker (cette classe de base pour les vérifications contient en particulier un mécanisme pour contrôler la profondeur des vérifications et éviter de contrôler plusieurs fois les mêmes objets). Toutes les fonctions membres qui commencent par check_ seront exécutées lors de l'instanciation de la classe SD que l'on vérifie. Il convient de noter que les deux obligations :

- la méthode doit obligatoirement commencer par check_
- la méthode doit avoir un objet checker en argument

La classe checker contient un dictionnaire de tous les objets JEVEUX déjà vérifiés, il suffit pour cela d'utiliser la donnée membre names :

```
    if checker.names.has_key(nomsd):  return
```

Cela se traduit par : si l'objet JEVEUX de nom nomsd a déjà été vérifié, alors return.

5.5 Accès au contenu des objets JEVEUX « get() »

Avec l'exemple précédent, nous avons introduit un autre mécanisme de contrôle, il s'agit de la ligne dime = self.DIME.get(). Il est en effet possible d'accéder au contenu des objets Fortran afin d'en récupérer des informations. Pour cela, le superviseur utilise les deux méthodes du module aster : getvectjev et getcolljev.

Il va de soi qu'il est tout à fait possible de définir des attributs et des méthodes spécifiques à la SD que l'on décrit. Par exemple, dans la sd_maillage, il existe une fonction membre u_dime donnant des informations génériques :

```
class sd_maillage(sd_titre):
  nomj   = SDNom(fin=8)
  DIME   = AsVI(lonmax=6, )
  def u_dime(self):
      dime=self.DIME.get()
      nb_no    =dime[0]
      nb_nl    =dime[1]
      nb_ma    =dime[2]
      nb_sm    =dime[3]
      nb_sm_mx =dime[4]
      dim_coor =dime[5]
      return nb_no, nb_nl, nb_ma, nb_sm, nb_sm_mx, dim_coor
```

Remarque : si l'objet est un objet simple, get() retourne une liste Python, si l'objet est une collection, get() retourne un dictionnaire Python.

5.6 Héritage et typage

Toutes les classes décrivant les SD peuvent être utilisées dans d'autres classes. Par exemple :

```
class sd_maillage(sd_titre):
   nomj = SDNom(fin=8)
   COORDO = sd_cham_no()
```

On voit dans cet exemple un double mécanisme. Le premier est l'héritage classique : la sd_maillage dérive de la sd_titre dont la description est :

```
class sd_titre(AsBase):
   TITR = AsVK80(SDNom(debut=19), optional=True)
```

La sd_titre ne contient qu'un vecteur de K80 stocké dans l'objet JEVEUX dont le nom commence au 19ème caractère. Cet objet est facultatif.

Le second mécanisme utilise le concept de typage des données propres à un langage objet comme Python. En effet, l'objet nomj(1:8)//'.COORDO' est une SD de type cham_no :

```
class sd_cham_no(sd_titre):
   nomj  = SDNom(fin=19)
   VALE = AsVect(ltyp=Parmi(4,8,16,24), type=Parmi('C', 'I', 'K', 'R'),
          docu=Parmi('', '2', '3'), )
   REFE = AsVK24()
   DESC = AsVI(docu='CHNO', )
```

Attention aux références circulaires (la SD maillage contient un objet cham_no qui contient un objet maillage). C'est au développeur d'y prendre garde (voir par exemple sd_cham_no).

5.7 Utilitaires

Un certain nombre d'opérations de vérification sont disponibles dans le module sd_util :

- sdu_assert(ojb, checker, bool,comment=) : vérifie que le booléen (bool) est vrai ;
- sdu_compare(ojb, checker, val1, comp, val2,) : vérifie que la relation de comparaison entre val1 et val2 est respectée avec comp= '==' / '!=' / '>=' / '>' / '<=' / '<' ;
- sdu_tous_differents(ojb,checker,sequence=None,) : vérifie que les éléments de la séquence sont tous différents ;
- sdu_tous_non_blancs(ojb,checker,sequence=None,) : vérifie que les éléments (chaines) de la séquence sont tous "non blancs" ;
- sdu_tous_compris(ojb,checker,sequence=None,vmin=None,vmax=None,) : vérifie que toutes les valeurs de la sequence sont comprises entre vmin et vmax ;
- sdu_monotone(seqini) : vérifie qu'une séquence est triée par ordre croissant (ou décroissant) ;
- sdu_nom_gd(numgd) : retourne le nom de la grandeur de numéro (numgd) ;
- sdu_licmp_gd(numgd) : retourne la liste des cmps de la grandeur de numéro (numgd) ;
- sdu_nb_ec(numgd) : retourne le nombre d'entiers codés pour décrire les composantes de la grandeur (numgd) ;
- sdu_ensemble(lojb) : vérifie que les objets JEVEUX de lojb existent simultanément.