Manuel de Descriptif Informatique
Fascicule D5.01 : -
Document : D5.01.02

Introduire une nouvelle macro-commande

Résumé

Ce document décrit comment définir et utiliser les macro-commandes en python.

1 Introduction

Ce document décrit l'utilisation et le développement des macro-commandes en Python pour *Code_Aster*. Ces macro-commandes peuvent être restituées dans le code et visibles de l'utilisateur comme des commandes à part entière. Mais elles peuvent aussi être placées dans le fichier de commande lui-même sans que l'utilisateur ait à toucher ni à l'exécutable, ni au catalogue de commandes. En outre, elles présentent l'avantage d'être écrites « naturellement » dans le langage de commande : le corps d'une macro-commande est semblable à un fichier de commandes ordinaire qui générerait la même séquence de commandes.

Le développeur aura grand intérêt à lire la programmation de macro-commandes existantes, sous `bibpyt/Macro` dans le répertoire d'installation de *Code_Aster*.

2 Qu'est-ce qu'une macro ?

Une macro est une commande qui regroupe l'exécution de plusieurs sous-commandes. Elle est utilisable dans un fichier de commandes comme toute autre commande et possède son propre catalogue, définissant sa syntaxe. Plusieurs types de macro sont possibles :

- des macros propres au superviseur, implémentées en Python et en Fortran : par exemple `FORMULE`, `INCLUDE`, `DEBUT` ...
- des macros développeurs implémentées en Python.

Ce document traite de la définition et de l'utilisation des macros en Python.

3 Utiliser les macros

Utiliser les macros en Python est simple. Par rapport à des commandes simples comme des `OPER` ou des `PROC`, la seule différence porte sur les concepts produits par la macro. Une commande simple, de type `OPER`, a un seul concept produit que l'on trouvera à gauche du signe « = », comme suit :

```
concept = commande(mots-cles-simples-ou-facteurs)
```

Une commande simple de type PROC n'a aucun concept produit et s'écrit :

```
commande(mots-cles-simples-ou-facteurs)
```

Une macro-commande, de type `MACRO`, peut avoir plusieurs concepts produits. Un que l'on trouvera à gauche du signe =, comme pour un `OPER`, les autres comme arguments des mots-clés simples ou facteurs. On présentera le mode d'emploi pour un mot-clé simple. Il s'étend facilement aux mots-clés facteurs. Certains mots-clés sont susceptibles de produire des concepts. Pour demander à une macro-commande de produire ce concept, l'utilisateur écrira à droite du signe = à la suite du nom du mot-clé, `CO('nom_concept')`, comme dans l'exemple qui suit :

```
MACRO_MATR_ASSE(NUMEDDL=CO('num'))
```

Ceci a pour effet de créer un concept produit de nom `num` en sortie de la commande `MACRO_MATR_ASSE`. Son type sera déterminé en fonction des conditions d'appel de la commande. `CO` est un nom réservé qui permet de créer des concepts produits nommés, non typés, préalablement à l'appel de la commande. C'est la commande qui attribuera le bon type à ce concept.

4 Définir une macro-commande en Python

Il faut définir :

- le catalogue proprement dit des mots-clés composant la macro,
- la méthode de typage des structures de données produites,
- la méthode Python définissant le corps de la macro : les commandes « de base » produites par la macro et leur enchaînement.

Les deux premiers points sont communs avec l'écriture d'une commande ordinaire (à l'exception d'une différence mineure dans la méthode de typage).

Il est possible de restituer tout ceci dans le catalogue de commandes du code et de rendre ainsi la macro-commande visible de tous. On peut également conserver son développement privé avec l'avantage de ne pas avoir à modifier les paramètres d'exécution ou l'exécutable en plaçant ces trois éléments directement en tête du fichier de commandes, ou en les important (import Python) depuis une localisation convenue.

4.1 Ecrire le catalogue de la macro-commande

Le catalogue d'une macro est semblable à celui d'un commande simple. Les trois différences sont :

- on déclare un objet `MACRO` (et non `PROC` ou `OPER`),
- le mot clé réservé `op` ne contient pas un entier (désignant le numéro de routine FORTRAN de haut niveau pour `OPER` et `PROC`) mais un nom de méthode python,
- le concept produit n'est pas nécessairement unique, déclaré à gauche du signe « = ».

Des concepts produits peuvent être spécifiés en tant qu'arguments d'un mot-clé simple. Si un mot-clé simple peut accepter un concept produit comme argument, il faut, en plus du type, spécifier `CO` dans le tuple `typ`.

Exemple de mot-clé simple acceptant un concept produit ou un concept existant :

```
NUME_DDL  =SIMP(statut='o',typ=(nume_ddl,CO))
```

Ici, le mot-clé accepte en argument un concept existant de type `nume_ddl` ou un concept à produire d'un type qui sera déterminé par la commande.

4.2 Définir le type des concepts produits

La définition du type des concepts produits d'une macro-commande s'effectue de manière semblable à celle d'une commande simple de type `OPER`.

Si la commande ne produit qu'un concept que l'on trouvera à gauche du signe = comme dans :

```
a  =COMMANDE()
```

on procédera de la même manière que pour une commande simple de type `OPER`.

Dans le cas où la macro-commande peut produire plusieurs concepts dont certains en arguments de mots-clés, il faut ajouter quelques informations supplémentaires. Tout d'abord, il faut absolument fournir une fonction Python, nommée `sd_prod`, dans la définition de la macro. Ensuite, les mots clés simples contenant le nom utilisateur du concept à produire doivent être de type `CO` (nom réservé).

Par exemple :

```
def ma_macro_prod(self,NUME_DDL,MATRICE,**args):
  if isinstance(NUME_DDL,CO) :
    self.type_sdprod(NUME_DDL,nume_ddl_sdaster)
  self.type_sdprod(MATRICE,matr_asse_depl_r)
  return evol_noli
....
MA_MACRO =MACRO(sd_prod=ma_macro_prod,...
             MATRICE  = SIMP(statut='o',typ=CO),
             ....
             NUME_DDL = SIMP(statut='o',typ=(CO,numeddl) ), )
```

Dans ce cas, trois concepts peuvent être produits :

- de façon classique, un concept de type `evol_noli` qui aura été donné par l'utilisateur à gauche du signe « = »
- un concept de type `matr_asse_depl_r` dont le nom aura été fourni par l'utilisateur derrière le mot clé simple `MATRICE`
- pour le cas `NUME_DDL`, l'utilisateur a ici le choix entre fournir un concept `numeddl` déjà existant ou le faire produire par la macro, auquel cas il détermine lui-même son nommage comme dans le cas `MATRICE`.

Lorsqu'un mot-clé peut avoir en argument un concept à produire, le mot-clé doit apparaître dans la liste des arguments de la fonction `sd_prod` et le concept doit être typé en utilisant la méthode `type_sdprod` de l'argument `self` qui est l'objet macro-commande.

NB : Cet argument self n'est pas présent pour un OPER ou une PROC.

4.3 Définir le corps de la macro

4.3.1 Transmission des mots clés de la macro à la méthode de construction (le corps)

Le corps de la macro sera défini dans une fonction dont les arguments sont semblables à ceux de la fonction `sd_prod`. Le premier argument est l'objet macro-commande, `self`, les suivants sont les mots-clés nécessaires pour exprimer le corps de la macro. Les mots-clés inutiles pour exprimer le corps de la macro seront ignorés par l'emploi de l'argument `**args`.

Seuls les mots clés de haut niveau sont transmis : les MCSIMP de premier niveau, les MCFACT. Ces mots clés sont ensuite invoqués très simplement par leur nom.

Exemple de fonction corps :

```
def ma_macro_ops(self,UNITE_MAILLAGE,**args):
   ..................
   _nomlma = LIRE_MAILLAGE( UNITE = UNITE_MAILLAGE )
   ..................
```

Ici, `UNITE_MAILLAGE` est un MCSIMP de la macro, son contenu (concept, liste, string, … peu importe) est affecté au MCSIMP `UNITE` de la commande `LIRE_MAILLAGE`.

Cas d'un mot clé facteur :

```
def ma_macro_ops(self,MATR_ASSE_GENE,**args):
   ..................
   if MATR_ASSE_GENE ['MATR_ASSE'] :
   ..................
```

MATR_ASSE_GENE est un MCFACT de la macro, MATR_ASSE est un de ses sous-MCSIMP. Un MCFACT se manipule comme un dictionnaire.

Astuce : dans ce dernier exemple, on teste très simplement la présence de MATR_ASSE : si l'utilisateur n'a pas renseigné un mot clé (simple ou facteur), il vaut par défaut None.

4.3.2 Appeler une commande dans le corps de la macro

Pour appeler une commande dans le corps de la macro, il est possible d'utiliser deux méthodes.

La première est la plus simple. Elle est possible si la commande existe dans le contexte global de la fonction définissant le corps de la macro. On se trouve dans ce cas au sein du catalogue de référence du code. On fera alors simplement :

```
num=NUME_DDL(METHODE=...,...)
```

Dans le cas où la commande n'existe pas dans le contexte, il faut interroger le catalogue par l'intermédiaire de la méthode get_cmd de l'objet macro-commande self pour obtenir cette commande :

```
NUME_DDL=self.get_cmd('NUME_DDL')
num=NUME_DDL(METHODE=...,...)
```

Il est indispensable d'utiliser le nom exact de la commande comme nom de la variable qui contiendra le retour de la méthode get_cmd. En effet ce nom est utilisé pour localiser la ligne de texte contenant ce nom et ainsi identifier le nom du concept produit (ici num).

Il n'y a aucun obstacle à ce que les commandes « filles » produites par la macro-commande soient elles-même des macro-commandes.

4.3.3 Relations entre concepts produits dans le corps et concepts produits par la macro

Les concepts produits dans le corps de la macro sont de plusieurs sortes :

- les concepts nommés automatiquement et détruits à la fin de l'exécution de la macro-commande. Il ne faut donc plus en avoir besoin par la suite et il faut en particulier veiller à ce que le concept produit par la macro n'y fasse pas référence. Pour indiquer qu'il s'agit d'un concept de cette sorte, il suffit de lui donner un nom qui commence par __ (double underscore)

  Exemple :

  ```
  __a=CALC_MATR_ELEM(MODELE=MODELE)
  ```

  Alors, comme on peut le lire dans le fichier de messages, un nom de concept automatique, précédé d'un point est généré :

  ```
  .9000005=CALC_MATR_ELEM(MODELE=MODELE)
  ```

  Une fois sorti de la macro, l'objet correspondant au nom de concept .9000005 n'existe plus, ni dans l'espace de noms du superviseur, ni dans la base jeveux.

- les concepts nommés automatiquement et conservés dans la base jeveux à la fin de la macro-commande. Pour indiquer qu'il s'agit d'un concept de cette sorte, il suffit de lui donner un nom qui commence par _ (simple underscore)

Exemple :

```
_a=CALC_MATR_ELEM(MODELE=MODELE)
```

Alors, comme on peut le lire dans le fichier de messages, un nom de concept automatique, précédé d'un underscore est généré :

```
_9000005=CALC_MATR_ELEM(MODELE=MODELE)
```

Une fois sorti de la macro, l'objet correspondant au nom de concept `_9000005` n'existe pas dans l'espace de noms du superviseur, il est en revanche présent sous ce nom dans la base `jeveux`.

Ce type d'objet répond aux situations particulières où le concept produit par la macro « dépend » d'un concept amont qui devra être toujours présent : par exemple un `modele` relativement à un `maillage`, une `matr_asse` relativement à un `nume_ddl`.

- les concepts destinés à devenir les concepts produits de la macro. Pour indiquer qu'il s'agit d'un concept de cette sorte, il faut appeler la méthode `DeclareOut` de l'objet macro `self` avec, comme arguments, le nom de la variable de retour de la commande et l'objet issu des mots-clés de la macro-commande.

  Exemple avec `matrice`, argument d'un mot-clé de la macro :

  ```
  self.DeclareOut('mm',matrice)
  mm=ASSE_MATRICE(.............)
  ```

- le concept de sortie de la macro (nécessairement unique) est traité de manière semblable en indiquant le concept de sortie `self.sd`.

  ```
  self.DeclareOut('mm',self.sd)
  mm=ASSE_MATRICE(.............)
  ```

  `mm` deviendra le concept de sortie de la macro, il portera le nom donné par l'utilisateur dans son fichier de commandes (et non `mm`).

4.3.4 Numéroter la macro

Dans les affichages du fichier de messages, toutes les commandes sont numérotées. Pour incrémenter ce compteur, il faut systématiquement appeler la méthode suivante en tête du corps de la macro :

```
self.set_icmd(1)
```

Il est particulièrement important de faire cet appel avant toute commande « fille » de la macro-commande car cette méthode initialise également la mesure de temps CPU globale pour la macro.

4.3.5 Traiter les erreurs

Comme pour une commande en Fortran, il est possible de détecter des erreurs d'utilisation dans le corps de la macro par l'utilitaire Utmess, identique dans son fonctionnement à son homonyme FORTRAN [D6.04.01]. Pour ce faire, il faut importer cette méthode depuis le module des utilitaires Python.

Exemple :

```
from Utilitai.Utmess     import UTMESS
...
message=        ' les deux embouts doivent etre \n'
message=message+' de meme longueur pour les cas de symetrie \n'
UTMESS('F', "MACR_ASCOUF_MAIL", message)
```

Le premier argument indique la nature de l'erreur ou alarme, le second précise la routine appelante et le dernier contient le message à destination de l'utilisateur.

4.3.6 Les affichages

Les affichages dans les fichiers de message et de résultat peuvent provenir de la partie programmée en Python comme de celle programmée en FORTRAN. Pour que le séquencement de ces affichages soit bien respecté (et donc que lesdits fichiers soient bien lisibles), il est fortement déconseillé d'utiliser la commande Python `print` mais plutôt employer l'utilitaire `affiche`, du module `aster`.

Exemple :

```
import aster
...
aster.affiche('MESSAGE',mon_texte_dans_une_string)
```

Le premier argument vaut `'MESSAGE'` ou `'RESULTAT'` suivant le fichier cible.

4.3.7 Identifier les concepts produits par la macro

Dans certaines circonstances, il faut déterminer si un concept est produit par la macro elle-même ou a été produit par une commande précédente. Ceci est possible en testant si le concept en question est présent ou non dans la liste des concepts produits par la macro qui est donnée par l'attribut `sdprods` de l'objet macro `self`.

Exemple :

```
if nume_ddl in self.sdprods :
  # le concept nume_ddl est produit par la macro
  # il faut appeler la commande NUME_DDL
  lnume = 1
else:
  # le concept nume_ddl existe déjà.
  lnume=0
```

Attention, `sdprods` ne contient pas le concept produit retourné par la macro qui se trouve dans l'attribut `sd` de `self`.

4.3.8 Creation dynamique de commandes : nombre de mots-clés variables, contenu contextuel

Dans certains cas, suivant la valeur des options, une même commande sera appelée avec différents mots-clés ou des arguments différents. Pour traiter cette situation et générer dynamiquement la commande, on construit un dictionnaire contenant les mots clés à écrire qui est ensuite transmis en argument de la commande, précédé des caractères '`**`'. Le dictionnaire est alors « déplié », les clés sont les arguments (mots clés), suivis du contenu de la clé, derrière le signe « = ».

Ce dictionnaire Python peut être construit au fur et à mesure de l'examen des options.

Exemple :

```
moscles={}
moscles['INFO'] = 2

if type('GROUP_MA_BORD')==types.StringType :
   motscles['CREA_GROUP_NO']  = _F(GROUP_MA =  GROUP_MA_BORD)
else :
   motscles['CREA_GROUP_NO']  = []
   for grma in GROUP_MA_BORD :
      motscles['CREA_GROUP_NO'] . append( _F(GROUP_MA = grma) )

_nomlma = DEFI_GROUP ( reuse = _nomlma
                       MAILLAGE = _nomlma,
                       **motscles )
```

La liste de MCFACT derrière la clé '`CREA_GROUP_NO`' du dictionnaire `motscles` contient un objet `_F` dans le premier cas et une liste de `_F` dans le second cas. La liste est construite par un `append` dans une boucle.

Remarque :

Dans cet exemple, si `CREA_GROUP_NO` *ne contient qu'un élément, ce n'est pas un singleton mais une string, d'où la nécessité d'importer le module* `types` *pour invoquer* `StringType` *ou* `ListType`.

4.3.9 Appel à un code externe

Si on souhaite exécuter dans la macro un code tiers par la commande `EXEC_LOGICIEL` ou par l'instruction python `os.system`, on doit récupérer le chemin des logiciels appelables dans une chaîne de caractères retournée par la routine FORTRAN repout.f ; cette routine est appelable depuis python par la méthode `repout` du module `aster`.

```
import os.path
import aster
chemin = aster.repout(  )
miss3d = os.path.join(chemin, 'miss3d')
EXEC_LOGICIEL( … , LOGICIEL  = miss3d ,  … )
```

5 Considérations sur l'usage des macros en Python

5.1 Définition d'une macro hors catalogue

La méthode standard pour ajouter la définition d'une macro en Python pour une exécution de *Code_Aster* est de l'ajouter dans le catalogue de référence du code.

Cependant, dans certains cas :

- macro personnelle,
- test lors du développement,

il peut être pratique d'ajouter la définition de la macro en dehors du catalogue. Pour ce faire, il suffit de créer un module Python contenant la définition de la macro en ajoutant en tête du module l'import des variables du catalogue.

Exemple simplifié :

```
from Cata.cata import *
def ma_macro_prod(self,...):
  .....
def ma_macro_ops(self,...):
  ....
MA_MACRO=MACRO(nom='MA_MACRO',...)
```

Puis à l'utilisation, il suffit dans le fichier de commandes de faire l'import de la macro précédemment définie.

Exemple de fichier de commandes :

```
# la macro MA_MACRO est définie dans le module ma_macro.py
from ma_macro import MA_MACRO
a=MA_MACRO(...)
```

Il est aussi possible d'utiliser la fonctionnalité d'`INCLUDE` :

```
# la macro MA_MACRO est définie dans le fichier INCLUDE 45
INCLUDE(UNITE=45)
a=MA_MACRO(...)
```

6 Quelques erreurs

Lors du développement ou de l'utilisation d'une macro-commande en Python, des erreurs d'utilisation ou de définition de la macro peuvent se produire. Les erreurs les plus caractéristiques sont présentées ci-dessous.

6.1 Argument de mot-clé invalide détecté dans le corps de la macro

Une éventuelle erreur utilisateur peut être détectée au niveau du corps de la macro-commande pendant la phase de construction des macros.

L'utilisateur obtiendra alors un compte-rendu de la forme :

```
JDC.py : Construction du JdC INVALIDE
CR de 1ere phase de construction de JDC
Etape : MA_MACRO    ligne : 42    fichier :
'/home01/chris/ASTER/bugs/Pymacros/tmp.1670/ahlv100a_err20'

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      ! LA PREMIERE OPTION DOIT ETRE RIGI_MECA OU RIGI_THER OU RIGI_ACOU OU
      ! RIGI_MECA_LAGR
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   Fin Etape : MA_MACRO
fin CR de 1ere phase de construction de JDC
```

6.2 Argument de mot-clé invalide détecté par une sous-commande

Une macro récupère les valeurs de ses mots-clés pour les transmettre à des commandes appelées sous-commandes de la macro. Il est possible que toutes les vérifications n'ayant pas été faites dans la définition de la macro, un argument de mot-clé de la sous-commande soit invalide. Cette erreur sera détectée au moment de la construction de la macro : méthode `Build` de l'objet jeu de commandes j du fichier JDC.py. L'utilisateur obtiendra un compte-rendu de la forme suivante :

```
JDC.py : Construction du JdC INVALIDE
DEBUT CR validation : SansNom
   Etape : MA_MACRO    ligne : 42    fichier :
'/home01/chris/ASTER/bugs/Pymacros/tmp.1677/ahlv100a_err10'
      Etape : NUME_DDL    ligne : 102    fichier : './ma_macro_err1.py'
         Mot-clé simple : METHODE
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
            ! toto n'est pas une valeur autorisée !
            !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! La valeur : 'toto'  n'est pas permise pour le mot-clé : METHODE !

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
         Fin Mot-clé simple : METHODE
      Fin Etape : NUME_DDL
   Fin Etape : MA_MACRO
FIN CR validation :SansNom
```

On peut voir que la commande `NUME_DDL` qui est une sous-commande de la macro `MA_MACRO` est invalide car le mot-clé `METHODE` est invalide.

6.3 Erreur de syntaxe dans la fonction corps de macro

Si une erreur de syntaxe est commise dans la fonction corps de macro, l'utilisateur obtiendra un compte-rendu de la forme suivante :

```
JDC.py : ERREUR ACCAS - INTERRUPTION
>> JDC.py : DEBUT RAPPORT
 CR phase d'initialisation
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! erreur non prevue et non traitee prevenir la maintenance ahlv100a_err4.py
!
! Traceback (most recent call last):                                       !
! File "./Eficas/Accas/commandes.py", line 2029, in __init__               !
! exec self.proc_compile in self.g_context                                 !
! File "/home01/chris/ASTER/bugs/Pymacros/tmp.1634/ahlv100a_err40", line 13, in ?
!
! from ma_macro_err4 import MA_MACRO                                        !
! File "./ma_macro_err4.py", line 46                                       !
! a=                                                                       !
! ^                                                                        !
! SyntaxError: invalid syntax                                              !
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
fin CR phase d'initialisation
>> JDC.py : FIN RAPPORT
```

6.4 Erreur de programmation dans la fonction corps de macro

Si une erreur de programmation est commise dans la fonction corps de macro, l'utilisateur obtiendra un compte-rendu de la forme suivante :

```
JDC.py : Construction du JdC INVALIDE
CR de 1ere phase de construction de JDC
   Etape : MA_MACRO    ligne : 41    fichier :
'/home01/chris/ASTER/bugs/Pymacros/tmp.1649/ahlv100a_err50'
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      ! impossible de construire la macro MA_MACRO                          !
      ! Traceback (most recent call last):                                  !
      ! File "./Eficas/Cata/asterexec.py", line 174, in Build_MACRO         !
      ! ier= apply(self.definition.proc,(self,),d)                          !
      ! File "./ma_macro_err5.py", line 44, in ma_macro_ops                 !
      ! a=1/0                                                               !
      ! ZeroDivisionError: integer division or modulo by zero               !
      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   Fin Etape : MA_MACRO
fin CR de 1ere phase de construction de JDC
```

6.5 Erreur de programmation dans la fonction sd_prod de la macro

Si une erreur de programmation est commise dans la fonction sd_prod de la macro, l'utilisateur obtiendra un compte-rendu de la forme suivante :

```
JDC.py : ERREUR ACCAS - INTERRUPTION
>> JDC.py : DEBUT RAPPORT

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
   ! Etape  MA_MACRO ligne :  41 fichier :                                   !
 ! /home01/chris/ASTER/bugs/Pymacros/tmp.1301/ahlv100a_err60  impossible d
affecter un type au resultat !
   ! File "./ma_macro_err6.py", line 10, in ma_macro_prod                    !
   ! a=1/0                                                                   !
   ! ZeroDivisionError: integer division or modulo by zero                   !

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
>> JDC.py : FIN RAPPORT
```

Important :

> *De plus l'interprétation du fichier de commandes est interrompue au niveau de l'erreur. En conséquence les erreurs éventuelles sur les commandes suivantes ne seront détectées que lors d'exécutions ultérieures*

7 Un exemple de macro-commande

On présente ci-dessous un exemple de définition et d'utilisation de macro-commande en Python. La macro-commande est définie dans un module Python de nom `ma_macro.py`. Ce module comprend 4 parties : un entête, une fonction `sd_prod` pour le typage des concepts produits [§7.1], une fonction pour la définition du corps de la macro [§7.2] et la définition des mots-clés de la macro [§7.3]. Cette macro reproduit presque complètement les fonctionnalités de la macro FORTRAN `MACRO_MATR_ASSE`.

Remarque :

> *Pour mieux comprendre les paragraphes [§7.1] et [§7.2], on pourra commencer par lire les paragraphes [§7.3] et [§7.4].*

7.1 Typage des concepts produits

```
# La fonction ma_macro_prod permet de définir le type de tous les concepts
# produits par la macro. Elle est identique à celle de MACRO_MATR_ASSE
def ma_macro_prod(self,NUME_DDL,MATR_ASSE,**args):
  if not MATR_ASSE:  raise AsException("Impossible de typer les concepts
resultats")
  if not NUME_DDL:  raise AsException("Impossible de typer les concepts
resultats")
  self.type_sdprod(NUME_DDL,nume_ddl)
  for m in MATR_ASSE:
    opti=m['OPTION']
    if opti in ( "RIGI_MECA","RIGI_FLUI_STRU","RIGI_MECA_LAGR" ,
       "MASS_MECA" , "MASS_FLUI_STRU" ,"RIGI_GEOM" ,"RIGI_ROTA",
       "AMOR_MECA","IMPE_MECA" ,
       "ONDE_FLUI","MASS_MECA_DIAG" ) : t=matr_asse_depl_r
    if opti == "RIGI_MECA_HYST"   : t= matr_asse_depl_c
    if opti == "RIGI_THER"        : t= matr_asse_temp_r
    if opti == "MASS_THER"        : t= matr_asse_temp_r
    if opti == "RIGI_THER_CONV"   : t= matr_asse_temp_r
    if opti == "RIGI_THER_CONV_D" : t= matr_asse_temp_r
    if opti == "RIGI_ACOU"        : t= matr_asse_pres_c
    if opti == "MASS_ACOU"        : t= matr_asse_pres_c
    if opti == "AMOR_ACOU"        : t= matr_asse_pres_c
    self.type_sdprod(m['MATRICE'],t)
```

```
  # La macro n'a pas de concept produit retourné à gauche du signe =.
  # Il faut retourner None
  return None
```

7.2 Corps de la macro

```
def ma_macro_ops(self,MODELE,CHAM_MATER,CARA_ELEM,MATR_ASSE,
                     SOLVEUR,NUME_DDL,CHARGE,INST,
                  **args):
  """
     Exemple de macro en Python reproduisant une partie des
     fonctionnalités de MACRO_MATR_ASSE
  """
  # Initialisation du compteur d'erreurs
  ier=0
  # import de l'utilitaire de messages d'erreurs
  from Utilitai.Utmess     import UTMESS
  nom_macro='MACRO_MATR_ASSE"
  # On met le mot cle NUME_DDL dans une variable locale pour le proteger
  numeddl=NUME_DDL

  # On importe les definitions des commandes a utiliser dans la macro
  # Le nom de la variable doit etre obligatoirement le nom de la commande
  CALC_MATR_ELEM=self.get_cmd('CALC_MATR_ELEM')
  NUME_DDL=self.get_cmd('NUME_DDL')
  ASSE_MATRICE=self.get_cmd('ASSE_MATRICE')

  # La macro compte pour 1 dans la numerotation des commandes
  self.set_icmd(1)

  if SOLVEUR:
    # Si le mot cle facteur SOLVEUR est present
    # On peut recuperer les valeurs des mots cles simples
    methode=SOLVEUR['METHODE']
    renum=SOLVEUR['RENUM']
  else:
    # Si le mot cle facteur SOLVEUR est absent
    # On affecte aux mots cles simples des valeurs par defaut
    methode='MULT_FRONT'
    renum='MDA'

  if methode == 'LDLT':
    if renum not in ('SANS','RCMK'):
      # Une erreur a été detectée. On incrémente le compteur et enregistre
      # un message fatal
      ier=ier+1
      UTMESS('F',nom_macro,"Avec methode LDLT, RENUM doit etre SANS ou
RCMK.")

  if numeddl in self.sdprods:
    # Si le concept numeddl est dans self.sdprods
    # il doit etre  produit par la macro
    # il faudra donc appeler la commande NUME_DDL
    lnume = 1
  else:
    lnume = 0

  iocc=0
  for m in MATR_ASSE:
    iocc=iocc+1
    option=m['OPTION']
    if iocc == 1 and lnume == 1 and option not in ('RIGI_MECA',
       'RIGI_MECA_LAGR','RIGI_THER','RIGI_ACOU'):
      ier=ier+1
      UTMESS('F',nom_macro," LA PREMIERE OPTION DOIT ETRE RIGI_MECA OU
RIGI_THER OU RIGI_ACOU OU RIGI_MECA_LAGR")
      # Une erreur a été detectée. On interrompt la construction de la
macro.
      return ier
```

```
      motscles={'OPTION':option}
      if CHAM_MATER != None: motscles['CHAM_MATER']=CHAM_MATER
      if CARA_ELEM != None: motscles['CARA_ELEM']=CARA_ELEM
      if CHARGE:
        if option not in ('RIGI_ACOU','MASS_ACOU'):
          motscles['CHARGE']=CHARGE
      if INST:motscles['INST']=INST
      if option == 'AMOR_MECA':
        motscles['RIGI_MECA']=rigel
        motscles['MASS_MECA']=masel
      sigg=m['SIEF_ELGA']
      if sigg:motscles['SIEF_ELGA']=sigg
      mh=m['MODE_FOURIER']
      if mh:motscles['MODE_FOURIER']=mh
          __a=CALC_MATR_ELEM(MODELE=MODELE,THETA=m['THETA'],
                             PROPAGATION=m['PROPAGATION'],**motscles)
      if option == 'RIGI_MECA':
        # On conserve dans rigel le resultat au cas ou
        rigel=__a
      if option == 'MASS_MECA':
        # On conserve dans masel le resultat au cas ou
        masel=__a
      if lnume and option in
  ('RIGI_MECA','RIGI_THER','RIGI_ACOU','RIGI_MECA_LAGR'):
        # On déclare que le concept produit de nom num est en réalité le
  concept numeddl
        self.DeclareOut('num',numeddl)
        # On peut passer des mots cles egaux a None. Ils sont ignores
        num=NUME_DDL(MATR_RIGI=__a,METHODE=methode,
                     RENUM=renum,TAILLE_BLOC=rbloc)
      else:
        num=numeddl
      self.DeclareOut('mm',m['MATRICE'])
      mm=ASSE_MATRICE(MATR_ELEM=__a,NUME_DDL=num)
    # On retourne le décompte des erreurs
    return ier
```

7.3 Définition des mots-clés

```
MA_MACRO = MACRO(nom="MA_MACRO",op=ma_macro_ops, sd_prod=ma_macro_prod,
      fr="Calcul des matrices assemblées (matr_asse_gd) par exemple de
rigidité, de masse ",
         MODELE          =SIMP(statut='o',typ=modele),
         CHAM_MATER      =SIMP(statut='f',typ=cham_mater),
         CARA_ELEM       =SIMP(statut='f',typ=cara_elem),
         CHARGE
=SIMP(statut='f',typ=(char_meca,char_ther,char_acou)),
         INST            =SIMP(statut='f',typ='R'),
         NUME_DDL        =SIMP(statut='o',typ=(nume_ddl,CO)),
         SOLVEUR         =FACT(statut='d',min=01,max=01,
           METHODE         =SIMP(statut='f',typ='TXM',defaut="MULT_FRONT",
                                 into=("LDLT","MULT_FRONT","GCPC")),
           TAILLE_BLOC     =SIMP(statut='f',typ='R'),

RENUM=SIMP(statut='f',typ='TXM',into=("SANS","RCMK","MD","MDA","METIS")),
         ),
         MATR_ASSE       =FACT(statut='o',min=01,max='**',
           MATRICE         =SIMP(statut='o',typ=(matr_asse,CO)),
           OPTION          =SIMP(statut='o',typ='TXM',

into=("RIGI_MECA","MASS_MECA","MASS_MECA_DIAG",

"AMOR_MECA","RIGI_MECA_HYST","IMPE_MECA",

"ONDE_FLUI","RIGI_FLUI_STRU","MASS_FLUI_STRU",

"RIGI_ROTA","RIGI_GEOM","RIGI_MECA_LAGR",

"RIGI_THER","MASS_THER","RIGI_ACOU","MASS_ACOU",
                                      "AMOR_ACOU")),
           SIEF_ELGA       =SIMP(statut='f',typ=cham_elem_sief_r),
           MODE_FOURIER    =SIMP(statut='f',typ='I'),
           THETA           =SIMP(statut='f',typ=theta_geom),
           PROPAGATION     =SIMP(statut='f',typ='R'),
         ),
         TITRE           =SIMP(statut='f',typ='TXM',max='**'),
         INFO            =SIMP(statut='f',typ='I',defaut=1,into=(1,2)),
)  ;
```

7.4 Utilisation de la macro

```
# Ce test d'utilisation est dérivé du cas test ahlv100a
# La macro MA_MACRO est définie dans le module privé ma_macro
# Il faut l'importer avec la commande Python import
from ma_macro import MA_MACRO
DEBUT(CODE=_F( NOM = 'AHLV100A') )
F=500.
MAIL=LIRE_MAILLAGE( )
AIR=DEFI_MATERIAU( FLUIDE=_F( RHO = 1.3, CELE_C = ('RI',343.,0.,)))
CHAMPMAT=AFFE_MATERIAU(  MAILLAGE=MAIL,
                                 AFFE=_F( TOUT = 'OUI',  MATER = AIR) )
GUIDE=AFFE_MODELE(  MAILLAGE=MAIL,     VERIF='MAILLE',
                           AFFE=_F( TOUT = 'OUI', MODELISATION = '3D',
                                    PHENOMENE = 'ACOUSTIQUE') )
CHARACOU=AFFE_CHAR_ACOU(      MODELE=GUIDE,
                  VITE_FACE=_F( GROUP_MA = 'ENTREE', VNOR =
('RI',0.014,0.,)))
IMPEACOU=AFFE_CHAR_ACOU(      MODELE=GUIDE,
                  IMPE_FACE=_F( GROUP_MA = 'SORTIE', IMPE =
('RI',445.9,0.,)))
# Si on compare l'utilisation de la macro MA_MACRO avec celle de
MACRO_MATR_ASSE
# on pourra constater qu'il n'y a aucune différence
MA_MACRO(
           MODELE=GUIDE, CHARGE=IMPEACOU,
```

```
                CHAM_MATER=CHAMPMAT,
                NUME_DDL=CO("NUM"),
                MATR_ASSE=(
                          _F(  MATRICE = CO("MATASK"), OPTION = 'RIGI_ACOU'),
                          _F(  MATRICE = CO("MATASM"), OPTION = 'MASS_ACOU'),
                          _F(  MATRICE = CO("MATASI"), OPTION = 'AMOR_ACOU'))
                        )
#
VECTELEM=CALC_VECT_ELEM(  OPTION='CHAR_ACOU',
                          CHAM_MATER=CHAMPMAT,
                          CHARGE=CHARACOU )
#
# IMPRESSION DU VECT_ELEM COMPLEXE VECTELEM SELON LE GRAIN MAILLE
#
IMPR_MATRICE(   MATR_ELEM=_F(  MATRICE = VECTELEM,
                               FORMAT = 'RESULTAT',
                               FICHIER = 'RESULTAT',
                               GRAIN = 'MAILLE'))
VECTASS=ASSE_VECTEUR(  VECT_ELEM=VECTELEM,   NUME_DDL=NUM )
#
#  ____________________________CALCUL DES MODES_____________________________________
#
# Comme pour une macro en Fortran, on utilise les concepts produits par
l'identificateur
# créé par la macro MATASK pour CO('MATASK') par exemple.
MATASKR=COMB_MATR_ASSE(    COMB_R=_F( MATR_ASSE = MATASK,
                                      PARTIE = 'REEL', COEF_R = 1.))
MATASMR=COMB_MATR_ASSE(    COMB_R=_F( MATR_ASSE = MATASM,
                                      PARTIE = 'REEL', COEF_R = 1.))
#
MODES=MODE_ITER_SIMULT(      MATR_A=MATASKR,
                             MATR_B=MATASMR,
                             CALC_FREQ=_F( OPTION = 'BANDE',
                                           NMAX_FREQ = 10,
                                           FREQ = ( 1., 1000.,))
                               )
```

Page laissée intentionnellement blanche.