Pour déboguer *Code_Aster*

Résumé :

Ce document a pour but de recenser les principaux outils qu'a à sa disposition le développeur *Code_Aster* pour :

- Déboguer un plantage ou un comportement anormal
- Détecter et éradiquer les écrasements, fuites et autres problèmes mémoires

Table des Matières

1 Déboguer le code

Le débogueur (*debugger*) est le principal outil et aussi le plus puissant dont dispose un développeur. Il permet de suivre en temps réel l'exécution d'un programme avec navigation interactive dans ses sources : on a ainsi la possibilité de faire avancer le code ligne à ligne, voire instruction par instruction, d'inspecter le contenu des variables et bien plus encore…

Pour déboguer, on utilise en général un exécutable compilé avec les symboles de débogage (ou de *debugging* accessibles par l'option `-g` sur la plupart des compilateurs). C'est l'exécutable produit par la commande `waf install_debug`. C'est aussi celui qui est utilisé lorsque l'on choisit `debug` dans `ASTK`.
L'ajout de ces symboles (et principalement la suppression des optimisations, équivalente au niveau `-O0`) produit en général un exécutable différent de celui de production, avec parfois une précision différente dans les calculs flottants.

Plusieurs modes d'utilisation existent :

- *Post-mortem :* quand il y a plantage, il est possible après une exécution, de remonter à l'endroit où celui-ci se produit grâce au "*core file*"
- Interactif : on lance l'exécutable "sous" le débogueur
- Couplage *a posteriori* : on connecte le débogueur à un programme en cours d'exécution

Le premier mode est utile lorsque l'on a besoin de connaître l'endroit exact d'un plantage mais que l'on ne veut pas ralentir outre-mesure l'exécution.

Le second mode est plus adapté lorsque l'on a déjà une idée du problème, puisque l'on va pouvoir aller examiner le contenu des objets et faire des vérifications élémentaires (on notera d'ailleurs que ce second cas est aussi utile lorsque l'on observe seulement un comportement anormal puisque l'on va pouvoir suivre l'état de certaines variables autrement que par des impressions).
Ce mode est celui de choix en général.

Enfin le dernier mode est utile lorsque que l'on se heurte à un problème de performances sur un calcul de taille conséquente, ou même lorsque qu'un calcul semble boucler indéfiniment. Il est alors difficile de réaliser un *profiling* [D1.06.01] : en connectant un débogueur à un programme en cours d'exécution, on peut examiner dans quelle routine il se trouve.

1.1 Post-mortem

En cas de plantage d'un calcul *Aster*, un débogueur est automatiquement exécuté en mode *post-mortem* pour donner des indications sur la localisation du plantage dans le source.
Le premier réflexe en cas de plantage est donc de relancer son calcul en mode « *debug* » pour obtenir une localisation précise (numéro de ligne dans le source de l'instruction illicite). Il faut cependant veiller à ce que les erreurs fatales provoquent un abandon du calcul (c'est-à-dire avoir renseigné le mot-clé `ERREUR=_F(ERREUR_F='ABORT')` dans `DEBUT`). C'est aussi le cas pour les cas-tests, la présence du mot-clé `CODE` active ce comportement en cas d'erreur.
Remarque : sur les plate-formes utilisant le compilateur Intel, on dispose directement du numéro de ligne en version optimisée (*nodebug*).

Si on veut aller plus loin, sans pour autant lancer son calcul sous le débogueur, on suivra les indications ci-dessous pour réaliser un débogage *post-mortem*.
Pour utiliser ce mode de débogage, il faut lancer son étude depuis `ASTK` en sélectionnant le mode « interactif » et en cliquant sur lancer "`pre`" par opposition à lancer "`run`".
`ASTK` prépare alors dans `/tmp` l'arborescence nécessaire au lancement d'*Aster* et indique dans le fichier d'*output* la ligne de commande à utiliser pour démarrer l'exécution après s'être placé au bon endroit.

On obtient le même fonctionnement en utilisant :
```
waf test_debug --name=zzzz000a --exectool=env
```
Comme avec ASTK, l'output indique les commandes à utiliser par la suite.

Exemple de l'output :
```
OK                Environnement de Code_Aster préparé dans /tmp/interactif.16468-dsp0764418

<INFO> Pour lancer l'exécution, copiez/collez les lignes suivantes dans un shell bash/ksh :
          cd /tmp/interactif.16468-dsp0764418
          . /xxx/public/v13/tools/Code_aster_frontend-salomemeca/etc/codeaster/profile.sh
          . /xxx/dev/codeaster/install/std/share/aster/profile.sh
          . profile_tmp.sh
<INFO> Ligne de commande 1 :
       cp fort.1.1 fort.1
       /xxx/dev/codeaster/install/std/bin/asterd
/xxx/dev/codeaster/install/std/lib/aster/Execution/E_SUPERV.py -commandes fort.1
--num_job=16468-dsp0764418 --mode=interactif
--rep_outils=/xxx/public/v13/tools/Code_aster_frontend-salomemeca/outils
--rep_mat=/xxx/dev/codeaster/install/std/share/aster/materiau
--rep_dex=/xxx/dev/codeaster/install/std/share/aster/datg --numthreads=1 --tpmax=60
--memjeveux=75.75
       Pour lancer l'exécution dans le debugger Python, vous pouvez utiliser :
       cp fort.1.1 fort.1
       /xxx/dev/codeaster/install/std/bin/asterd /usr/lib/python2.7/pdb.py
/xxx/dev/codeaster/install/std/lib/aster/Execution/E_SUPERV.py -commandes fort.1
--num_job=16468-dsp0764418 --mode=interactif
--rep_outils=/xxx/public/v13/tools/Code_aster_frontend-salomemeca/outils
--rep_mat=/xxx/dev/codeaster/install/std/share/aster/materiau
--rep_dex=/xxx/dev/codeaster/install/std/share/aster/datg --numthreads=1 --tpmax=60
--memjeveux=75.75
```

Pour le débogage "post-mortem" 3 étapes sont nécessaires :

1) Positionner l'environnement d'exécution (recopier depuis l'output les lignes ad-hoc, il y a plus ou moins de lignes selon l'environnement à positionner) :

```
cd /tmp/interactif.16468-dsp0764418
. /xxx/public/v13/tools/Code_aster_frontend-
salomemeca/etc/codeaster/profile.sh
. /xxx/dev/codeaster/install/std/share/aster/profile.sh
. profile_tmp.sh
```

2) Exécuter le code en interactif (recopier depuis l'output les lignes ad-hoc) :

```
ulimit -c unlimited
cp fort.1.1 fort.1

/xxx/dev/codeaster/install/std/bin/asterd
/xxx/dev/codeaster/install/std/lib/aster/Execution/E_SUPERV.py -commandes fort.1
--num_job=16468-dsp0764418 --mode=interactif
--rep_outils=/xxx/public/v13/tools/Code_aster_frontend-salomemeca/outils
--rep_mat=/xxx/dev/codeaster/install/std/share/aster/materiau
--rep_dex=/xxx/dev/codeaster/install/std/share/aster/datg --numthreads=1 --tpmax=60
--memjeveux=75.75
```

La première commande permet de s'assurer que le *corefile* pourra être écrit sans limite de taille, sinon il est possible qu'il ne soit pas produit du tout.

3) Lancer le débogueur "*post-mortem*" :

Lorsque le calcul a planté, le système a produit un fichier que l'on appelle *core file*. Ce fichier qui contient l'état de la mémoire au moment du plantage permet l'analyse *post-mortem*. Attention : si le programme utilisait une grande quantité de mémoire au moment du plantage, ce fichier peut être volumineux.
Ce fichier est nommé `core` ou parfois `core.NNN` où `NNN` est un numéro.

L'analyse *post-mortem* est réalisée en lançant :
```
gdb /xxx/dev/codeaster/install/std/bin/asterd core
```

La première instruction exécutée dans le débogueur est en général `where` pour savoir où s'est arrêté le programme !
Pour ce qui concerne la navigation une fois le débogueur lancé, on se reportera à la section suivante.

1.2 Débogage interactif

1.2.1 Fonctionnement général

On peut également déboguer le code de façon plus interactive en lançant l'exécution de *Code_Aster* sous le contrôle d'un débogueur. Un débogueur est un outil permettant l'avancée d'un programme ligne à ligne et l'examen de toutes les variables rencontrées dans le source.
Un tel outil rend de nombreux services et peut s'avérer extrêmement puissant. Les débogueurs couramment utilisés sont *gdb* (GNU, en mode texte, fonctionne partout) ou *idb* (Intel) . En général, on utilisera une interface graphique qui est plus conviviale que ces simples outils en ligne de commande : on peut citer DDD, nemiver (interfaces à *gdb*) ou bien IDB (interface Eclipse à *idb*).

Concrètement, pour lancer l'exécution de *Code_Aster* sous le contrôle du débogueur, il faut utiliser le bouton "`Lancer / dbg`" de `ASTK`. La version exécutée est alors automatiquement la version *debug*. L'interface graphique se lance, elle positionne immédiatement un premier point d'arrêt qui a pour effet de stopper le programme dans le *main* du programme `python.c` (le point d'entrée de *Code_Aster*).
L'équivalent avec `waf` est obtenu en exécutant :

```
waf test_debug --name=zzzz000a --exectool=debugger
```

Si le débogueur ne se lance pas ou pour changer de débogueur, voir le paragraphe 1.5 Configurer le débogueur.

Avant de poursuivre l'exécution, on peut positionner d'autres points d'arrêts. Une fois ceci terminé, on continue l'exécution en appuyant sur « *cont* » ou « *run* » selon l'interface graphique utilisée.

Quelques commandes de `gdb` (syntaxe très proche pour `idb`)

- Aide en ligne : « man `gdb` » ou bien dans `gdb` taper `help`, ou `help` *`subject`*
- Touche « `ENTER` » reproduit l'action précédente
- Où suis-je ? : `where`, `up`, `down` (permet de se déplacer dans la pile d'appels)
- Spécifier un point d'arrêt dans une routine ou à une ligne donnée de la routine courante :
  `break` *`nom_routine`*
  `break` *`num_ligne`*
  exemple : `break op0199` ou `break 87` ou `b op0199`
  exemple 2 : `break 87 if (i .eq. 3)` (on s'arrête à ligne 87 du fichier courant si la variable locale *i* vaut 3)
  Dans certains débogueurs, `break` est remplacé par `stop in`/`stop at`.
- Continuer l'exécution jusqu'au point d'arrêt suivant : `cont` ou `c`
- Lister les points d'arrêt : `info breakpoints` ou `status`
- Détruire un point d'arrêt : `delete` *`id`*
- Désactiver un point d'arrêt : `disable` *`id`*
- Avancer d'une instruction en restant dans la routine en cours : `next` ***ou*** `n`
- Avancer d'une instruction en plongeant dans les routines appelées : `step` ***ou*** `s`
- Afficher le contenu d'une variable : `print` *`nom_var`* ou `p` *`nom_var`*
- Affiche une expression à chaque arrêt : `display nom_var` ou `display expression`
- Remplir une variable : `set` *`nom_var=valeur`*
- Lister le programme : `list` ou `list` *`num_ligne`*
- Tuer le programme courant : `kill` , pour le redémarrer : `run`
- Sauvegarder les points d'arrêts pour réutilisation : `save breakpoints` *`filename`*

Toutes ces commandes ont en général un équivalent graphique (bouton) ou bien un raccourci (par exemple dans *idb* ce sont les touches fonctions).

1.2.2 Débogage d'un programme parallèle avec Totalview

Lorsque le calcul à déboguer est parallèle (MPI par exemple), il est possible d'utiliser un débogueur dédié comme Totalview qui se chargera de lancer le calcul parallèle et donnera accès à la position de chaque processus MPI.
La démarche pour lancer un calcul *Aster* sous Totalview diffère quelque peu du mode interactif séquentiel.

Les étapes à suivre sont les suivantes :

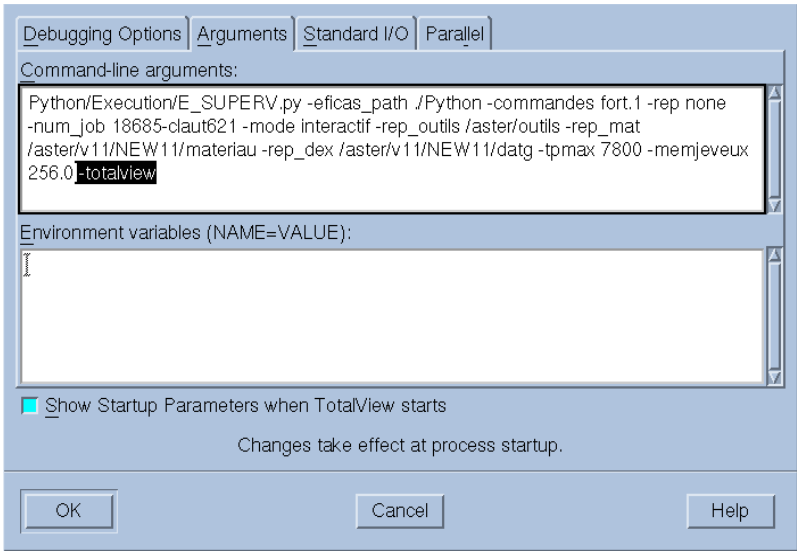
- préparation d'un répertoire temporaire d'exécution avec le mode « *pre* »
- positionnement de l'environnement et lancement de Totalview
- paramétrage de Totalview

Pour la première étape, on se reportera au §1.1. On prendra garde à sélectionner **la version parallèle et un seul processeur**.
Dans la seconde étape, le positionnement de l'environnement et le lancement de Totalview se font à l'aide du fichier d'*output* produit par la première étape :

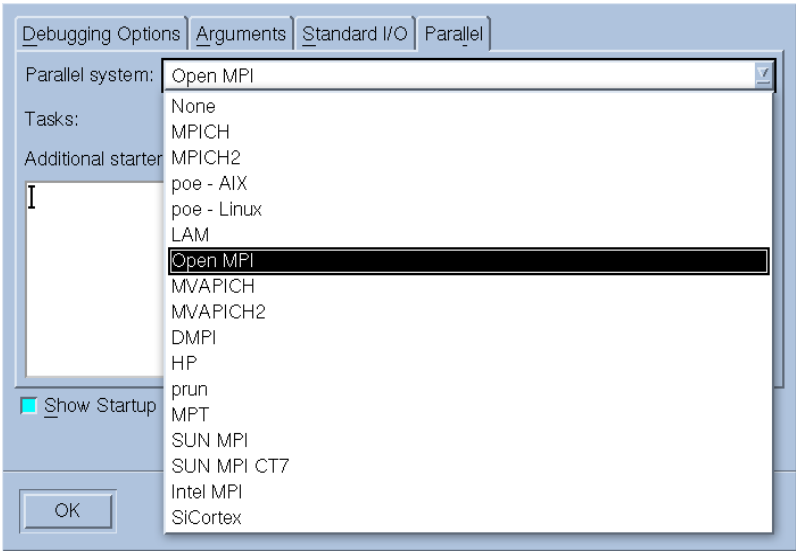
```
cd /tmp/interactif.16468-dsp0764418
. /xxx/public/v13/tools/Code_aster_frontend-salomemeca/etc/codeaster/profile.sh
. /xxx/dev/codeaster/install/std/share/aster/profile.sh
. profile_tmp.sh

cp fort.1.1 fort.1
totalview /xxx/dev/codeaster/install/std/bin/asterd
```

Le paramétrage de Totalview se fait de façon similaire aux deux images ci-dessous. On notera l'argument « -totalview » rajouté à la suite des arguments donnés dans le fichier d'output.



Le pilote MPI à utiliser peut différer selon les plateformes. Le menu « Tasks » permet de préciser le nombre de processus à lancer.



1.3 Débogage d'un programme en cours d'exécution

1.3.1 Introduction

Parce qu'il n'est parfois pas possible de réaliser un *profiling*, on souhaite interrompre un programme pour savoir où il passe le plus clair de son temps, ou tout simplement où il semble boucler. Il est bien évidemment possible de surcharger le code pour y placer des impressions mais cela demande de connaître *a priori* l'endroit du blocage ou bien de travailler par dichotomie ce qui peut devenir long (si le calcul en question est une étude).

On propose ici une technique très simple utilisant un débogueur (`gdb`).

1.3.2 Mise en application sur un exemple

Considérons le calcul suivant :

```
[desoza@aster3 ~]$ date && bjobs
Tue Jun 30 09:39:35 CEST 2009
JOBID  USER   STAT  QUEUE      FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
721238 desoza  RUN  q16G_24h   aster2       aster10   *_gros_cas Jun 29 12:59
```

Il tourne depuis 20h40min. Si on regarde dans son répertoire d'exécution :

```
[desoza@aster3 ~]$ ssh aster10
Last login: Tue Jun 30 08:51:25 2009 from aster3
cd[desoza@aster10 ~]$ cd /tmp/721238
[desoza@aster10 721238]$ ls -ltrh
total 1.5G
-rwxrwxr-x   1 desoza astergrp  81M Jun 24 00:42 asteru
-rw-r--r--   1 desoza astergrp    2 Jun 29 12:59 msg_job
-rw-r--r--   1 desoza astergrp   12 Jun 29 12:59 FTMPDIR
-rw-r--r--   1 desoza astergrp    0 Jun 29 12:59 fort.0
drwxr-xr-x   2 desoza astergrp    6 Jun 29 13:00 RESU_ENSIGHT
drwxr-xr-x   2 desoza astergrp    6 Jun 29 13:00 REPE_OUT
drwxr-xr-x   2 desoza astergrp   35 Jun 29 13:00 rep_coco
-rw-r--r--   1 desoza astergrp  852 Jun 29 13:00 721238.export
-rwxr-xr-x   1 desoza astergrp 8.1M Jun 29 13:00 fort.20
-rw-r--r--   1 desoza astergrp    0 Jun 29 13:00 err_cp
-rwxr-xr-x   1 desoza astergrp 7.9K Jun 29 13:00 fort.1
-rw-r--r--   1 desoza astergrp    0 Jun 29 13:00 err
drwxr-xr-x  17 desoza astergrp 4.0K Jun 29 13:00 Eficas
-rw-r--r--   1 desoza astergrp 6.5K Jun 29 13:00 config.txt
-rw-r--r--   1 desoza astergrp  595 Jun 29 13:01 fort.9
-rwxr-xr-x   1 desoza astergrp  15M Jun 29 13:01 elem.1
-rw-r--r--   1 desoza astergrp 8.3K Jun 29 13:03 fort.8
-rw-r--r--   1 desoza astergrp  61K Jun 29 13:03 fort.6
-rw-r--r--   1 desoza astergrp 245M Jun 29 13:04 glob.1
-rw-r--r--   1 desoza astergrp 778K Jun 29 13:04 fort.15
-rw-r--r--   1 desoza astergrp 1.1G Jun 29 13:04 vola.1
```

On constate que le calcul n'a rien écrit sur disque depuis 20h35min. De fait il n'a même pas fini une itération de Newton :

```
                    …
                    CARA_ELEM=springs,
                    MODELE=tipo,
                    CHAM_MATER=fisica,
                    );

 --- NOMBRE TOTAL DE NOEUDS :                 80435  DONT :
               25030  NOEUDS "LAGRANGE"
 --- NOMBRE TOTAL D'EQUATIONS :                191245
 --- TAILLE DU PROFIL MORSE DE LA TRIANGULAIRE SUPERIEURE (FORMAT SCR):      3740478
 --- DONC LA TAILLE DE LA MATRICE EST:
 --- EN SYMETRIQUE     NNZ=      3740478
 --- EN NON SYMETRIQUE NNZ=      7289711

 --- NOMBRE TOTAL DE NOEUDS ESCLAVES :                  5369


-------------------------------------------------------------------------------------------------------------------------

INSTANT DE CALCUL :  2.000000000E-02

-------------------------------------------------------------------------------------------------------------------------
|     CONTACT     |   ITERATIONS    |     RESIDU      |     RESIDU      |     OPTION      |     CONTACT     |    REAC GEOM    |
|                 |     NEWTON      |     RELATIF     |     ABSOLU      |   ASSEMBLAGE    |    DISCRET      |    MAXIMUM      |
|   ITER. GEOM.   |                 | RESI_GLOB_RELA | RESI_GLOB_MAXI |                 |   ITERATIONS    |                 |
-------------------------------------------------------------------------------------------------------------------------
```

Nous allons utiliser les fonctionnalités de `gdb` (ou tout autre *debugger*) qui permettent d'interrompre un programme après s'y être attaché. Il va pour cela falloir connaître le `PID` de l'exécutable Aster qui tourne en boucle. On peut par exemple utiliser la commande suivante (ne fonctionne que si l'exécutable en question s'appelle "`asteru`" et qu'il n'y a qu'un seul *job* au nom de l'utilisateur sur le nœud de calcul) :

```
[desoza@aster10 721238]$ pgrep -u $USER asteru
2595
```

Lorsque le job que l'on veut ausculter est en parallèle, il est délicat de trouver le `PID` du processeur `i`. Une possibilité est d'utiliser l'outil "`top`", d'afficher les colonnes `PID` et `PPID` (Parent `PID`) et de remonter le numéro du processus "`asteru`" à partir du numéro du répertoire temporaire de la forme "`proc_pid`" (ici `pid` est le `PID` du script *shell* de lancement du calcul *Aster*). L'idée est que l'on cherche dans la colonne `PPID`, le numéro `pid`, on trouve alors dans la colonne `PID` un nouveau numéro que l'on cherche à nouveau dans la colonne `PPID`, et ainsi de suite jusqu'à arriver au processus "`./asteru ...`".

Nous exécutons ensuite la ligne suivante après s'être placé dans le répertoire temporaire d'exécution :

```
[desoza@aster10 721238]$ gdb ./asteru 2595
```

Cela donne la chose suivante :

```
GNU gdb Bull Linux (6.3.0.0-1.132.EL4.b.2.Bull)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "ia64-redhat-linux-gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

Attaching to program: /tmp/721238/asteru, process 2595
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xa000000000000000
`shared object read from target memory' has disappeared; keeping its symbols.
Reading symbols from /opt/intel/cmkl/9.1.023/lib/64/libmkl.so...done.
Loaded symbols for /opt/intel/cmkl/9.1.023/lib/64/libmkl.so
…

Loaded symbols for /aster/local/Python-2.4.5/lib/python2.4/lib-dynload/_random.so
Reading symbols from /aster/local/Python-2.4.5/lib/python2.4/lib-dynload/md5.so...done.
Loaded symbols for /aster/local/Python-2.4.5/lib/python2.4/lib-dynload/md5.so
Reading symbols from /opt/intel/cmkl/9.1.023/lib/64/libmkl_i2p.so...done.
Loaded symbols for /opt/intel/cmkl/9.1.023/lib/64/libmkl_i2p.so
0x4000000000358a20 in tldlr8_??unw ()
```

Nous avons ainsi interrompu le programme (il n'est plus dans l'état `'running' R` mais dans l'état `'stopped' T`) comme le montre l'outil *top*.

```
  PID  PPID USER      VIRT SWAP  RES CODE DATA  P S %CPU %MEM    TIME+  nFLT COMMAND
 2595  2546 desoza   7351m 127m 7.1g  64m 7.0g  2 T  0.0  5.5   1245:46    6 ./asteru
```

On peut désormais faire comme dans un *debugger*, et demander où on est pour savoir ce qui se passe :

```
(gdb) where
#0  0x4000000000358a20 in tldlr8_??unw ()
#1  0x40000000000355ee0 in tldlg3_??unw ()
#2  0x4000000000357860 in tldlgg_??unw ()
#3  0x400000000256b2b0 in algoco_??unw ()
#4  0x40000000023e2f10 in cfalgo_??unw ()
#5  0x40000000022575c0 in nmcofr_??unw ()
#6  0x4000000001c48210 in nmcoun_??unw ()
#7  0x4000000001046480 in nmdepl_??unw ()
#8  0x40000000004044e0 in op0070_??unw ()
....
```

Comme on utilise une version "`nodebug`" on n'a pas accès au source (et aux numéros de ligne), il faudrait pour cela une version compilée avec l'option "`-g`". Néanmoins on peut déterminer ce qui se passe. Ici il s'agit d'un calcul de contact qui mouline dans `tldlr8` la routine qui factorise le complément de Schur du contact. Comme il y a plus de 4000 nœuds de contacts actifs, cette factorisation est très longue (mais c'est normal).

Lorsque l'on a fini dans `gdb`, on peut se détacher du programme puis quitter, l'exécution reprend alors son cours.

```
(gdb) detach
Detaching from program: /tmp/721238/asteru, process 2595
(gdb) q
```

1.4 Déboguer le source Python

Lorsque le *bug* concerne les sources Python, il faut utiliser le débogueur Python. Pour cela, on utilise encore ASTK / Lancer "pre". Après avoir fait quelque chose comme :

```
cd /tmp/interactif.12219
export ASTER_VERSION=NEW9
. /opt/aster/ASTK/ASTK_SERV/conf/aster_profile.sh
. /opt/aster/NEW9/profile.sh
```

On peut lancer le code sous le contrôle du débogueur Python :

```
To start execution in the Python debugger you could type :
./asterd /usr/lib/python2.7/pdb.py Python/Execution/E_SUPERV.py -eficas_path \
./Python -commandes fort.1.1 -rep none  -num_job 12219 -mode interactif \
 -rep_outils /opt/aster/outils -rep_mat /opt/aster/NEW9/materiau \
 -rep_dex /opt/aster/NEW9/datg -tpmax 120 -memjeveux 16.0
```

Pour plus de détails, voir par exemple : http://docs.python.org/library/pdb.html

1.5 Configurer le débogueur

La ligne de commande utilisée pour exécuter le débogueur (en mode interactif ou *post-mortem*) est définie dans les fichiers de configuration de ASTK.
Pour voir la ligne de commande utilisée en interactif, faire :

```
as_run --showme param cmd_dbg
```

Pour voir la ligne de commande utilisée en *post-mortem*, faire :

```
as_run --showme param cmd_post
```

En général, ces commandes sont définies dans le fichier de configuration du serveur.
Vous pouvez modifier cette valeur en écrivant la ligne de commande de votre choix dans $HOME/.astkrc/prefs.

Attention

Le fichier à modifier est $HOME/.astkrc_salomemeca_VERSION/prefs si ASTK est issu de Salome-Meca.

Exemple pour *gdb*, copier/coller la ligne :

```
echo 'cmd_dbg : xterm -e gdb --command=@D @E @C' >> ~/.astkrc/prefs
```

Exemple pour *nemiver*, copier/coller la ligne :

```
echo 'cmd_dbg : nemiver @E @a' >> ~/.astkrc/prefs
```

Exemple pour *idb*, copier/coller la ligne :

```
echo 'cmd_dbg : /opt/intel/Compiler/11.1/064/bin/ia32/idb -gui -gdb
-command @D -exec @E' >> ~/.astkrc/prefs
```

Les codes @E, @C, ... sont remplacés par ASTK au moment du lancement :

- @E : nom de l'exécutable Code_Aster,
- @a : les arguments passés à l'exécutable Code_Aster,
- @C : nom du corefile,
- @D : nom du fichier de commandes pour le débogueur (qui contient where + quit),
- @d : le texte correspondant aux commandes passées au débogueur,

2 Valgrind

2.1 Présentation

Valgrind est un exécutable permettant de détecter certaines erreurs de programmation lors de l'exécution d'un programme. Le principe de fonctionnement de Valgrind est de surcharger certaines fonctions systèmes. Ceci est fait au travers d'une bibliothèque dynamique et des fonctions telles que *malloc, free*, *memcpy* sont ainsi remplacées par des équivalents instrumentés par Valgrind.

Pour en savoir plus : http://valgrind.org/docs/manual/quick-start.html

ou bien :

```
man valgrind
valgrind --help
```

2.2 Utilisation

Pour analyser un calcul avec Valgrind, l'exécutable *Aster* doit être compilé avec les symboles de débogage (version "debug" à cocher dans ASTK).

Exemple d'usage (pour vérifier le programme unix "ls") :

```
valgrind --tool=memcheck --error-limit=no  ls
```

Plus généralement, une bonne ligne de commande Valgrind ressemble à :

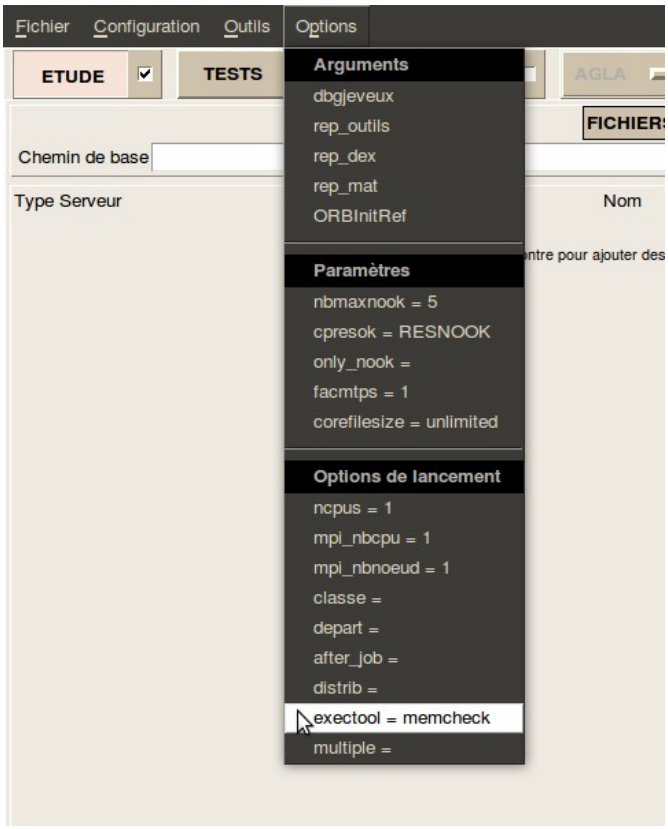
```
valgrind --tool=memcheck --error-limit=no --leak-check=full \
--suppressions=python.supp --track-origins=yes
```

Le fichier python.supp permet de supprimer les erreurs Python non justifiées (Python dispose de son propre gestionnaire de mémoire qui se permet des manipulations non standards). On trouve en général un exemplaire de ce fichier dans les distributions Linux. Sur Calibre, celui-ci se trouve dans /usr/lib/valgrind/python.supp.

Pour utiliser Valgrind avec *Aster*, il faut pouvoir "encapsuler" l'appel à l'exécutable. Cette technique "d'encapsulation" peut se faire de plusieurs façons mais on ne détaille ici que la plus simple (et celle qui est recommandée).
On utilise pour cela la fonctionnalité "exectool" de ASTK. On commence par renseigner dans son fichier local de configuration (situé dans $HOME/.astkrc/prefs) des alias vers des lignes de commandes qui préfixeront la ligne de lancement d'*Aster :*

```
desoza@claut621:~$ echo 'memcheck : valgrind --tool=memcheck --error-limit=no --leak-check=full
--suppressions=/chemin/vers/python.supp --track-origins=yes' >> ~/.astkrc/prefs
```

Ensuite dans le menus "Options", on déclare exectool=memcheck. Puis on lance le calcul normalement. Un message s'affiche alors pour confirmer que l'on veut lancer le calcul avec l'outil sélectionné.



L'équivalent avec `waf` est obtenu en exécutant (modifier le `.export` du test pour augmenter la limite en temps) :

```
waf test_debug --name=zzzz000a --exectool=memcheck
```

Plusieurs remarques peuvent être faites :

- L'exécution sous `valgrind` peut être beaucoup plus longue (30 fois plus parfois). Il est préférable d'utiliser un exécutable "`debug`" pour que le diagnostic `valgrind` soit plus précis (numéro de ligne dans les sources). Penser donc à allouer suffisamment de temps dans `ASTK`. Il est aussi parfois nécessaire d'augmenter la limite mémoire sous peine d'obtenir un arrêt brutal en cours d'exécution sans information claire.
- Avec l'option `--num-callers=n`, on choisit la profondeur *n* de l'arbre d'appel affiché par `Valgrind`.
- L'option *`--track-origins=yes`* n'est disponible qu'à partir des versions de Valgrind supérieures à 3.4.0.

2.3 Décryptage

Une fois le calcul lancé, les messages d'erreur détectés par Valgrind se retrouveront alors mélangés à l'*output* d'*Aster*. Ils sont signalés par des "`==NumeroDeProcessus==`" et on a en général 3 types d'erreurs possibles :

- Utilisation d'une variable non initialisée
- Lecture invalide en dehors d'un segment mémoire
- Écriture invalide en dehors d'un segment mémoire

Variable non initialisée

```
==8906==
==8906== Conditional jump or move depends on uninitialised value(s)
==8906==    at 0x9167E47: nbsuco_ (nbsuco.f:124)
==8906==    by 0x90459F2: poinco_ (poinco.f:130)
==8906==    by 0x8E5CB3F: limaco_ (limaco.f:120)
==8906==    by 0x8C0AAC7: calico_ (calico.f:284)
==8906==    by 0x8BEE78F: charme_ (charme.f:194)
==8906==    by 0x833047F: op0007_ (op0007.f:66)
==8906==    by 0x81D82B9: ex0000_ (ex0000.f:69)
==8906==    by 0x8175A10: execop_ (execop.f:83)
==8906==    by 0x81028F6: expass_ (expass.f:82)
==8906==    by 0x80CDBDD: aster_oper (astermodule.c:2621)
==8906==    by 0x408288C: PyCFunction_Call (in /usr/lib/libpython2.5.so.1.0)
==8906==    by 0x40D05E8: PyEval_EvalFrameEx (in /usr/lib/libpython2.5.so.1.0)
```

Dans le cas de variables initialisées, il est possible si le problème ne saute pas aux yeux de demander à `Valgrind` de remonter la chaîne et d'indiquer dans quelle routine la variable non initialisée a été créée. Il faut pour cela rajouter l'option "`-track-origins=yes`". Cette option est disponible à partir de la version 3.4.0.

Lecture ou écriture invalide

```
==11092==
==11092== Invalid write of size 4
==11092==    at 0x94894EE: ajellt_ (ajellt.f:327)
==11092==    by 0x9426F37: cazocc_ (cazocc.f:552)
==11092==    by 0x93863C2: cazoco_ (cazoco.f:170)
==11092==    by 0x90DC5A3: caraco_ (caraco.f:93)
==11092==    by 0x8C0DF43: calico_ (calico.f:279)
==11092==    by 0x8BF2FB7: charme_ (charme.f:194)
==11092==    by 0x832CE8B: op0007_ (op0007.f:66)
==11092==    by 0x81D942D: ex0000_ (ex0000.f:69)
==11092==    by 0x8176114: execop_ (execop.f:83)
==11092==    by 0x81031F2: expass_ (expass.f:82)
==11092==    by 0x80CE40D: aster_oper (astermodule.c:2621)
==11092==    by 0x408288C: PyCFunction_Call (in /usr/lib/libpython2.5.so.1.0)
==11092==  Address 0x5D3A8E4 is 0 bytes after a block of size 40,036 alloc'd
==11092==    at 0x4022765: malloc (vg_replace_malloc.c:149)
==11092==    by 0x816B470: hpalloc_ (hpalloc.c:30)
==11092==    by 0x80FAA8B: jjalls_ (jjalls.f:113)
==11092==    by 0x8126D61: jxveuo_ (jxveuo.f:231)
==11092==    by 0x80FDE70: jjalty_ (jjalty.f:59)
==11092==    by 0x8104CE1: jeveuo_ (jeveuo.f:142)
==11092==    by 0x94876F6: ajellt_ (ajellt.f:114)
==11092==    by 0x9426F37: cazocc_ (cazocc.f:552)
==11092==    by 0x93863C2: cazoco_ (cazoco.f:170)
==11092==    by 0x90DC5A3: caraco_ (caraco.f:93)
==11092==    by 0x8C0DF43: calico_ (calico.f:279)
==11092==    by 0x8BF2FB7: charme_ (charme.f:194)
```

Ce bloc se présente en deux parties. La partie haute donne la description de l'erreur et sa localisation dans le source. Ici dans ajellt.f à la ligne 327, on fait une écriture de 4 octets en dehors du segment mémoire qui avait été alloué. Pour information la ligne ressemblait à cela :

```
ZI(IDLITY+ZI(IDPOMA+ZI(IDAPMA)-1)+I-1) = ITYP
```

La partie basse donne l'origine du problème. En effet on apprend que l'on se situe à l'adresse `0x5D3A8E4` avec un décalage de 0 octets par rapport au segment mémoire dans lequel on est en train d'écrire (autrement dit on est au bout de ce segment). On comprend donc bien que si l'on fait une écriture de taille 4 octets, on sort du segment mémoire. L'information la plus précieuse du bloc `Valgrind` est que l'objet en dehors duquel on écrit a été alloué dans ajellt.f à la ligne 114.

```
CALL JEVEUO(LIGRET//'.LITY','E',IDLITY)
```

En regardant les attributs de l'objet `LIGRET.LITY`, on s'aperçoit qu'il était dimensionné en dur à une longueur 1000, d'où le problème.

2.4 Erreurs détectées par valgrind mais que l'on peut « oublier »

Il est admis que les erreurs de type « `Conditional jump or move depends on uninitialised value(s)` » détectées sur les routines suivantes ne sont pas problématiques :

- `jjcrec.f`
- `codree.f`

2.5 Valgrind pour les nuls

Pour lancer une étude avec `valgrind`

1) vérifier que `as_run -showme param memcheck` retourne bien une ligne pour exécuter valgrind.
2) Multiplier le temps de l'étude par 100 dans `astk`
3) dans `astk/options/exectool` écrire `memcheck`
4) lancer l'étude en `debug`

Analyse du fichier .mess

1) rechercher les occurrences de « `conditional jump` »
2) si la dernière routine fortran dans la remontée ne fait pas partie de la liste des routines exemptées (voir §2.4) alors il y a un vrai problème : une variable non initialisée est déclarée dans cette routine. Pour pister cette variable `VAR`, on peut rajouter des `IF (VAR.EQ.XX)` dans le source.

3 Débogage JEVEUX

L'usage de JEVEUX peut conduire à certaines erreurs particulières.

Deux outils permettent au développeur de détecter ces erreurs :

- le mode d'exécution en "debug jeveux"
- la routine jxveri.f

3.1 "debug jeveux"

Le mode d'exécution "debug jeveux" s'active dans ASTK en cochant Options/Paramètres/dbgjeveux avant de lancer l'exécution.

Le code s'exécute alors (plus lentement) en provoquant systématiquement la lecture et/ou l'écriture des objets JEVEUX lorsqu'ils sont demandés ou libérés de la mémoire (routines jeveuo, jelibe, jedema). De plus, quand un objet jeveux est détruit (jedetr, jedetc), la zone mémoire qu'il occupait est mise à "undef". Ce comportement du code permet de provoquer une erreur d'exécution quand :

- On continue à utiliser un objet après sa destruction
- On continue à utiliser un objet qui a été "libéré"
- On écrit dans un objet alors qu'on a demandé un accès en "lecture".

3.2 JXVERI

jxveri.f est la subroutine de *Code_Aster* permettant de détecter un écrasement dans la mémoire statique de JEVEUX.

Elle est utile lorsque le code s'arrête avec l'un des messages d'erreur suivants :

```
JEVEUX_15 : Ecrasement amont ...
JEVEUX_16 : Ecrasement aval ...
JEVEUX_17 : Chainage cassé …
```

L'objet du débogage est alors de localiser l'instruction qui provoque l'écrasement de la mémoire JEVEUX. Pour cela, on agit par itérations successives.

1ère étape
On localise la commande coupable en utilisant **DEBUT(DEBUG=_F(JXVERI='OUI'))**

2ème étape
On surcharge (en mode debug) la routine op00ij correspondant à la commande coupable en la "truffant" de **call jxveri(' ',' ')** :

```
subroutine op00ij(...)
        ...
        call jxveri(' ',' ')
        bloc 1
        call jxveri(' ',' ')
        bloc 2
        call jxveri(' ',' ')
        bloc 3
        call jxveri(' ',' ')
end
```

Lors de l'exécution du code ainsi surchargé, le code s'arrêtera en erreur fatale au 1er appel à `jxveri` coupable. Si par exemple, il a lieu à la fin du bloc 2 (on connait la ligne grâce au "`traceback`" imprimé par le débogueur "*post-mortem*"), alors, on réitère le processus en ajoutant des "`call jxveri`" entre les différentes instructions du bloc 2. Et ainsi de suite ...

Dans la pratique, le processus converge assez rapidement vers l'instruction fautive.

4 Autres outils

Dans cette partie, on décrit quelques outils qui permettent aussi de trouver des *bugs* ou de débusquer des comportements anormaux :

- Option "`-Checkbounds`" des compilateurs
- Comparaison de 2 versions différentes de *Code_Aster*

4.1 Dépassement de tableaux (`-CheckBounds`)

Les compilateurs disposent d'outils permettant d'instrumenter le code pour détecter des dépassements de tableaux statiques (un des types de *bugs* difficiles à trouver en Fortran).

Pour utiliser ces fonctionnalités, il faut recompiler les routines suspectes avec ces options (il faut pour cela modifier le `config.txt` et le mettre en Donnée dans l'onglet Surcharge) puis exécuter le code. Si un dépassement survient, une erreur fatale avec un message se produira.

```
Syntaxe :
Gcc (g77) : -fbounds-check
Intel (ifort) : -CB
```

Remarques :

Les routines utilisant les COMMON `JEVEUX ( ZI , ZR , ...)` ne peuvent pas être compilées avec -CB car alors l'exécution s'arrête rapidement du fait du débordement du tableau ZI(1) .
Du coup, l'utilisation de -CB est un peu compliquée : il faut jongler avec 2 fichiers config.txt et conserver les fichiers .o .
L'intérêt de -CB n'est pas énorme car ce mécanisme ne détecte pas tous les écrasements de tableaux. Pour que l'écrasement soit détecté, il faut que le tableau soit local (donc dimensionné en "dur"), ou bien que ce soit un tableau argument déclaré avec sa longueur exacte (et nom pas `TAB(*)`). L'autre intérêt de -CB est la détection des écrasements des chaines de caractères car en fortran la longueur d'une chaine est attachée à la chaine. C'est pour cela que l'on peut faire len(chaine) sur une chaine que l'on a reçue en argument (alors que l'on ne peut pas faire len(TAB)).

```
Gcc (g77) : -fbounds-check

-fbounds-check
    -ffortran-bounds-check
        Enable generation of run-time checks for array subscripts and substring start
        and end points against the (locally) declared minimum and maximum values.

    The current implementation uses the "libf2c" library routine "s_rnge" to print the
    diagnostic.

    However, whereas f2c generates a single check per reference for a multi-
    dimensional array, of the computed offset against the valid offset range (0
    through the size of the array), g77 generates a single check per sub- script
    expression.  This catches some cases of potential bugs that f2c does not, such as
    references to below the beginning of an assumed-size array.

    g77 also generates checks for "CHARACTER" substring references, something f2c
    currently does not do.

    Use the new -ffortran-bounds-check option to specify bounds-checking for only the
    Fortran code you are compiling, not necessarily for code written in other
    languages.

    Note: To provide more detailed information on the offending subscript, g77
    provides the "libg2c" run-time library routine "s_rnge" with somewhat differently-
    formatted information.  Here's a sample diagnostic:
    Subscript out of range on file line 4, procedure rnge.f/bf.
    Attempt to access the -6-th element of variable b[subscript-2-of-2].
    Aborted

    The above message indicates that the offending source line is line 4 of the file
    rnge.f, within the program unit (or statement function) named bf.  The offended
    array is named b.  The offended array dimension is the second for a two-
    dimensional array, and the offending, computed subscript expression was -6.

    For a "CHARACTER" substring reference, the second line has this appearance:Attempt
    to access the 11-th element of variable a[start-substring].

    This indicates that the offended "CHARACTER" variable or array is named a, the
    offended substring position is the starting (leftmost) position, and the offending
    substring expression is 11.

    (Though the verbage of "s_rnge" is not ideal for the purpose of the g77 compiler,
    the above information should provide adequate diagnostic abilities to it users.)

    Some of these do not work when compiling programs written in Fortran:


Intel (ifort) : -CB

-CB    Performs run-time checks on whether array subscript and
       substring references are within declared bounds
       (same as the -check bounds option).
```

Exemple de détection d'erreur :

```
forrtl: severe (408): fort: (2): Subscript #1 of the array RESU
has value 4 which is greater than the upper bound of 3

Image              PC                Routine           Line     Source
asteru_jpl         0000000001F9E956  Unknown           Unknown  Unknown
asteru_jpl         0000000001F9DB56  Unknown           Unknown  Unknown
asteru_jpl         0000000001F11232  Unknown           Unknown  Unknown
asteru_jpl         0000000001EDA0E2  Unknown           Unknown  Unknown
asteru_jpl         0000000001ED9068  Unknown           Unknown  Unknown
asteru_jpl         00000000004933AE  mkkvec_                52  mkkvec.f
asteru_jpl         0000000000493AD0  mmmab2_                40  mmmab2_jpl.f
asteru_jpl         0000000000E6A639  te0364_               370  te0364.f
asteru_jpl         0000000000910304  te0000_              1261  te0000.f
asteru_jpl         0000000000613998  calcul_               472  calcul.f
asteru_jpl         0000000000EE74A0  mmcmat_               149  mmcmat.f
asteru_jpl         0000000000D9F12F  mmcmem_                69  mmcmem.f
asteru_jpl         00000000009D308F  nmdepl_               300  nmdepl.f
asteru_jpl         00000000007A4EA8  op0070_               304  op0070.f
asteru_jpl         0000000000599772  ex0000_               258  ex0000.f
asteru_jpl         00000000004E6750  execop_                90  execop.f
asteru_jpl         00000000004D26EE  expass_                82  expass.f
asteru_jpl         0000000000499BD7  aster_oper           2635  astermodule.c
```

4.2 Comparaison de 2 versions différentes de *Code_Aster*

Il arrive parfois que deux exécutions différentes de *Code_Aster* conduisent à des résultats différents.

Cela peut se produire :

- Avec la même version du code sur deux plateformes différentes.
- Avec deux versions différentes (`N` et `N+1`) sur la même plateforme
- Avec la même version mais avec les deux exécutables "`debug`" et "`nodebug`"
- ...

Le problème à résoudre est alors d'identifier le morceau de code qui a un comportement différent pour les deux exécutions. Pour localiser le problème, on peut déclencher des impressions intermédiaires à quelques endroits "stratégiques" du code :

- lors de chaque appel aux calculs élémentaires (routine calcul.f)
- lors de chaque appel à la routine de résolution de système linéaire (routine resoud.f)

En faisant un `diff` (ou un `tkdiff`) sur les 2 fichiers message produits, on peut localiser l'endroit où les 2 versions divergent.

Mise en œuvre

Pour déclencher ces impressions, il faut surcharger la routine calcul.f et/ou resoud.f. On modifie alors le source en forçant la variable : DBG=.TRUE..

Cela entraine alors des impressions supplémentaires dans le fichier message.

routine `calcul.f`

Par exemple, les impressions de la routine `calcul.f` lors du calcul de l'option AMOR_ACOU sont :

```
1 &&CALCUL|IN  |PGEOMER | MAIL    .COORDO    .VALE | LONMAX= ...| SOMMR= 0.58898033E+03
2 &&CALCUL|IN  |PIMPEDC | IMPEACOU.CHAC.IMPED.VALE | LONMAX= ...| SOMMR= 0.13370000E+04
3 &&CALCUL|IN  |PMATERC | CHAMPMAT.MATE_CODE .VALE | LONMAX= ...| SOMMI= 743107436
4 &&CALCUL OPTION=AMOR_ACOU       ACOU_FACE8               182
5 &&CALCUL|OUTG|PMATTTC | _9000024.ME001     .RESL | LONMAX= ...| SOMMR= 0.74828831E-04
6 &&CALCUL|OUTF|PMATTTC | _9000024.ME001     .RESL | LONMAX= ...| SOMMR= 0.74828831E-04
```

Les lignes 1,2,3 correspondent aux 3 paramètres "in" de cette option. Pour chaque paramètre, on imprime des informations sur le champ associé à ce paramètre : nom du champ, LONMAX de l'objet contenant les valeurs du champ, ... et "résumé" (colonne SOMMR ou SOMMI) des valeurs du champ.

La ligne 4 indique que le ligrel sur lequel est fait le calcul contient un grel d'éléments de type ACOU_FACE8 et que la routine te00ij.f appelée est le te0182.f
La ligne 5 renseigne sur le champ "out" PMATTTC après les calculs élémentaires du grel ACOU_FACE8 (donc du te0182.f).

Les lignes 4 et 5 peuvent être répétées s'il y a plusieurs grel dans le ligrel.

La ligne 6 renseigne sur le champ "out" après le calcul de tous les grel.

Il peut arriver que les impressions montrent que bien que les champs "in" d'un calcul élémentaire soient identiques, les champs "out" diffèrent. On sait alors que le problème concerne un calcul élémentaire précis : OPTION type_element et numéro de la routine te00ij.f.

Remarques :

Lorsqu'un champ est de type entier, réel ou complexe, le nombre résumant le champ (SOMMI ou SOMMR) est un nombre obtenu en "sommant" les valeurs du champ. En réalité, un léger "biais" est introduit pour permettre la détection d'une permutation des valeurs : le vecteur (1 2 3 4) conduira en général à un SOMMI différent de (2 3 1 4).
Pour les champs de type CHARACTER , on fait une somme entière (SOMMI) en transformant chaque caractère en entier (fonction ICHAR).
Attention : un champ "in" est presque toujours différent entre deux exécutions, c'est le champ de "matériau codé" ('PMATERC') : il contient des adresses JEVEUX qui n'ont aucune raison d'être identiques. D'autres objets JEVEUX ont également presque toujours un contenu différent à chaque exécution, ce sont les objets .TITR qui contiennent en général la date de l'exécution.
Détail : Pour chaque objet JEVEUX "résumé", on imprime : son nom, sa "somme" (SOMMI ou SOMMR) son LONMAX, son LONUTI, son TYPE (R/C/I/K8/...), un code_retour IRET (si iret /= 0, l'objet JEVEUX est dans un état douteux) ainsi qu'un nombre IGNORE qui compte les valeurs "ignorées" dans la somme (SOMMI ou SOMMR). Les valeurs ignorées sont les valeurs 'Nan' ou invalides (pour faire la somme) : R8MAEM(), R8VIDE(), ISMAEM(), ...

routine `resoud.f`

Les impressions de la routine `resoud.f` sont :

```
1 &&RESOUD 2ND MEMBRE | &&MESTAT.2NDMBR_ASS.VALE | LONMAX= ... | SOMMR= -0.20000000000E+06
2 &&RESOUD CHCINE     | &&ASCAVC.VCI       .VALE | LONMAX= ... | SOMMR=  0.00000000000E+00
3 &&RESOUD MATR.VALM  | &&MESTAT_MATR_ASSEM.VALM | LONMAX= ... | SOMMR=  0.13926619473E+13
4 &&RESOUD MATR.VALF  | &&MESTAT_MATR_ASSEM.VALF | LONMAX= ... | SOMMR=  0.12301036782E+13
5 &&RESOUD MATR.CONL  | &&MESTAT_MATR_ASSEM.CONL | LONMAX= ... | SOMMR=  0.55076923235E+12
6 &&RESOUD SOLU       | &&MERESO_SOLUTION  .VALE | LONMAX= ... | SOMMR= -0.37488024534E+06
```

Ligne 1 : second membre du système linéaire
Ligne 2 : valeurs des degrés de liberté imposés éliminés (char_cine)
Ligne 3 : valeurs de la matrice initiale (avant factorisation)
Ligne 4 : valeurs de la matrice factorisée
Ligne 5 : valeur du coefficient de conditionnement des Lagranges (ddls imposés dualisés)
Ligne 6 : valeurs de la solution

Si la ligne 1 diffère, le problème provient de la fabrication du second membre du système.
Si la seule ligne 4 diffère, cela traduit un problème de factorisation (routine `preres.f`)
Si la seule ligne 6 diffère, le problème vient de la résolution (routine `resoud.f`).