To debug *Code_Aster*

Summary:

The purpose of this document is to count the principal tools which at its disposal the developer has *Code_Aster* for:

- To debug a planting or an abnormal behavior
- To detect and eradicate crushings, escapes and other problems reports

Contents

1 To debug the code

The debugger (*débuguer*) is the principal tool and also most powerful available to a developer. It makes it possible to follow in real time the execution of a program with interactive navigation in its sources: there is thus the possibility of advancing the code line with line, even instruction by instruction, to inspect the contents of the variables and much more still…

To debug, one in general uses achievable compiled with the symbols of debugging (or of *debugging* accessible by the option - G on most compilers). It is the achievable one produced by the order waf install_debug. It is also that which is used when one chooses debug in ASTK.
The addition of these symbols (and mainly suppression of optimizations, equivalent to the level - O0) achievable product in general different from that of production, with sometimes a different precision in floating calculations.

Several modes of use exist:

- *Post-mortem:* when there is planting, it is possible after an execution, to go back to the place where this one occurs thanks to "*core file*"
- Interactive: one launches the achievable one "under" the debugger
- Coupling *a posteriori* : one connects the debugger to a program in the course of execution

The first mode is useful when one needs to know the exact place of a planting but which one does not want to slow down in addition to-measurement the execution.

The second mode is adapted more when one has already an idea of the problem, since one will be able to go to examine the contents of the objects and to make elementary checks (one will note besides that this second case is also useful when one observes only one abnormal behavior since one will be able to differently follow the state of certain variables than by impressions).
This mode is that of choice in general.

Finally the last mode is useful when that one encounters a problem of performances on a calculation of consequent size, or even when that a calculation seems to buckle indefinitely. It is then difficult to carry out one *profiling* [D1.06.01]: by connecting a debugger to a program in the course of execution, one can examine in which routine it is.

1.1 Post-mortem

In the event of planting of a calculation *Aster*, a debugger is automatically carried out in mode *post-mortem* to give indications on the localization of planting in the source.
The first reflex in the event of planting is thus to start again its calculation in mode " *debug* " to obtain a precise localization (number of line in the source of the illicit instruction). It should however be taken care that the fatal errors cause an abandonment of calculation (i.e. to have informed the keyword ERREUR=_F (ERREUR_F=' ABORT') in BEGINNING). It is also the case for the CAS-tests, the presence of the keyword CODE activate this behavior in the event of error.
Note: on platforms using the Intel compiler, one has directly the number of line in version optimized (*nodebug*).

If one wants to go further, without to launch his calculation under the debugger, one will follow the indications below to carry out a debugging *post-mortem*.
To use this mode of debugging, its study should be launched since ASTK by selecting the "interactive" mode and while clicking on launching "pre" in opposition to launching "run".
ASTK prepare then in /tmp the tree structure necessary to the launching of*Aster* and indicates in the file of*output* the command line to use to start the execution after being itself placed at the good place.

One obtains same operation while using:

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

As with ASTK, the output indicates the orders to be used thereafter.

Example of the output:

```
 OK                Environment of Code_Aster prepared in /tmp/interactif.16468-dsp0764418

 <INFO> to launch the execution, copy/stick the following lines in 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> Command line 1:
        CP fort.1.1 fort.1

 /xxx/dev/codeaster/install/std/bin/asterd/xxx/dev/codeaster/install/std/lib/aster/Executio
 n/E_SUPERV.py - orders 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
        To launch the execution in the débuguer Python, you can use:
        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 - orders fort.1
 --num_job=16468-dsp0764418 --mode=interactif
 --rep_outils=/xxx/public/v13/tools/Code_aster_frontend-salomemeca/outils
 --rep_mat=/xxx/dev/codeaster/installation/standard/share/aster/materiau
 --rep_dex=/xxx/dev/codeaster/installation/standard/share/aster/datg --numthreads=1
 --tpmax=60 --memjeveux=75.75
```

For debugging “post-mortem” 3 stages are necessary:

1) To position the environment of execution (to recopy since the output the ad hoc lines, there are more or less lines according to the environment to position):

```
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) To carry out the code in interactive (to recopy since the output the ad hoc lines):

```
ulimit - C unlimited
CP fort.1.1 fort.1

/
xxx/dev/codeaster/install/std/bin/asterd/xxx/dev/codeaster/install/std/lib/aster/Exec
ution/E_SUPERV.py - orders 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
```

The first order allows to make sure that it *corefile* could be written unbounded of size, if not it is possible that it is not produced a whole.

3) To launch the debugger “*post-mortem*”:

When calculation planted, the system produced a file which one calls *core file*. This file which contains the state of the memory at the time of planting allows the analysis *post-mortem*. Caution: if the program used a great quantity of memory at the time of planting, this file can be bulky.
This file is named `core` or sometimes `core.NNN` where `NNN` is a number.

Analysis *post-mortem* is realized while launching:
`gdb/xxx/standard Dev./codeaster/installation//bin/asterd core`

The first instruction carried out in the debugger is in general `where` to know where the program stopped!

Concerning navigation once the launched debugger, one will refer to the following section.

1.2 Interactive debugging

1.2.1 Operation general

One can also debug the code in a more interactive way while launching the execution of *Code_Aster* under the control of a debugger. A debugger is a tool allowing the projection of a program line line and the examination of all the variables met in the source.
Such a tool does many favours and can prove extremely powerful. The debuggers usually used are *gdb* (GNU, in text mode, functions everywhere) or *idb* (Intel) . In general, one will use a graphic interface which is more convivial than these simple tools on command line: one can quote DDD, Nemiver (interfaces with *gdb*) or IDB (interface Eclipses with *idb*).

Concretely, to launch the execution of *Code_Aster* under the control of the debugger, it is necessary to use the button "`To launch/dbg`" of `ASTK`. The version carried out is then automatically the version *debug*.
The graphic interface launches out, it immediately positions a first stagnation point which causes to stop the program in *hand* program `python.c` (the entrance point of *Code_Aster*).
The equivalent with `waf` is obtained while carrying out:

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

If the debugger does not launch out or to change debugger, to see the paragraph 1.5 To configure the debugger.

Before continuing the execution, one can position other stagnation points. Once this finished, one continues the execution while supporting on " *cont* " or " *run* " according to the graphic interface used.

Some orders of `gdb` (very close syntax for `idb`)

- Online help: "man `gdb` " or in `gdb` to type `help`, or `help` *`subject`*
- Touch " `ENTER` " reproduced the preceding action
- Where am I? : `where`, `up`, `down` (allows to move in the pile of calls)
- To specify a stagnation point in a routine or with a given line of the current routine:
  `station-wagon` *`nom_routine`*
  `station-wagon` *`Num_line`*
  example: `station-wagon op0199` or `station-wagon 87` or `B op0199`
  example 2: `station-wagon 87 yew (I.eq. 3)` (one stops with line 87 of the file running if the local variable *I* is worth 3)
  In certain debuggers, `station-wagon` is replaced by `stop in`/`stop At`.
- To continue the execution up to the following stagnation point: `cont` or `C`
- To list the stagnation points: `information breakpoints` or `status`
- To destroy a stagnation point: `delete` *`id`*
- To disable a stagnation point: `sayable` *`id`*
- To advance of an instruction while remaining in the routine in progress: `next` *or* `N`
- To advance of an instruction while plunging in the routines called: `step` *or* `S`
- To display the contents of a variable: `print` *`nom_var`* or `p` *`nom_var`*
- Poster an expression with each stop: `display nom_var` or `display expression`
- To fill a variable: `set` *`nom_var=valor`*
- To list the program: `list` or `LSTI` *`num_ligne`*
- To kill the program running: `bottle` , to start again it: `run`
- To save the stagnation points for re-use: `save breakpoints` *`filename`*

All these orders in general have a graphic equivalent (button) or a short cut (for example in *idb* they are them keys functions).

1.2.2 Debugging of a parallel program with Totalview

When calculation to be debugged is parallel (MPI for example), it is possible to use a dedicated debugger as Totalview which will be given the responsibility to launch parallel calculation and will give access to the position of each process MPI.
The approach to launch a calculation *Aster* under Totalview differs somewhat from the sequential interactive mode.

The stages to be followed are the following ones:

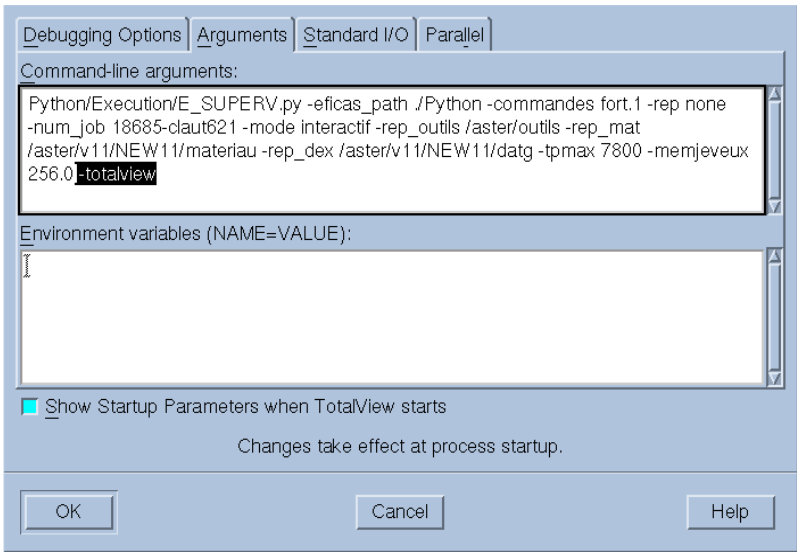
- preparation of a temporary repertoire of execution with the mode " *pre* "
- positioning of the environment and launching of Totalview
- parameter setting of Totalview

For the first stage, one will refer to the §1.1. One will take care to be selected **the parallel version and only one processor**.
In the second stage, the positioning of the environment and the launching of Totalview are done using the file*output* product by the first stage:

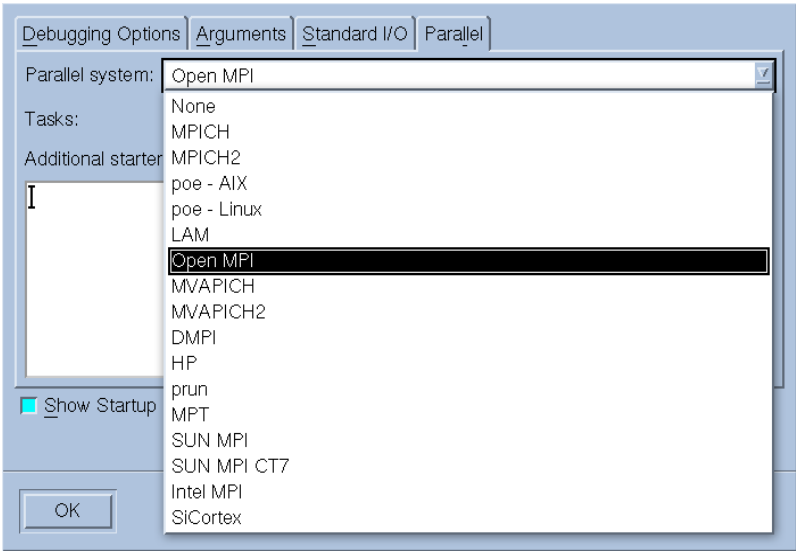
```
Cd /tmp/interactif.16468-dsp0764418
. /xxx/public/v13/tools/Code_aster_frontend-salomemeca/etc/codeaster/profile.sh
. /xxx/dev/codeaster/install/std/share/aster/profile.sh
. profile_tmp.sh

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

The parameter setting of Totalview is made in a way similar to the two images below. One will note the argument "- totalview" added following the arguments given in the file of output.



Pilot MPI to be used can differ according to the platforms. Small "the Task" makes it possible to specify the number of processes to launching.



1.3 Debugging of a program in the course of execution

1.3.1 Introduction

Because it is sometimes not possible to carry out one *profiling*, one wishes to stop a program to know where it spends most clearly its time, or quite simply where it seems to buckle. It is obviously possible to overload the code to place there impressions but that requires to know *a priori* the place of blocking or to work by dichotomy what can become long (if calculation in question is a study).

One proposes a very simple technique here using one debugger (`gdb`).

1.3.2 Application on an example

Let us consider following calculation:

```
[desoza@aster3 |] $ date && bjobs
Kill Jun 30 09:39: 35 IT IS 2009
JOBID  TO USE   STAT  TAIL     FROM_HOST   EXEC_HOST   JOB_NAME   SUBMIT_TIME
721238 desoza  RUN  q16G_24h   aster2      aster10   *_gros_cas Jun 29 12:59
```

It turns since 20:40 min. If one looks in his repertoire of execution:

```
 [desoza@aster3 |] $ HS aster10
 Last login: Kill 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    June 2nd, 29 12:59 msg_job
 - rw-r--R--   1 desoza astergrp   June 12th, 29 12:59 FTMPDIR
 - rw-r--R--   1 desoza astergrp    0 Jun 29 12:59 fort.0
drwxr-xr-x   2 desoza astergrp    June 6th, 29 13:00 RESU_ENSIGHT
drwxr-xr-x   2 desoza astergrp    June 6th, 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 Eficas 13:00
 - 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
```

It is noted that calculation did not write anything on disc since 20:35 min. In fact it did not even finish an iteration of Newton:

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

 --- FULL NUMBER OF NODES:                 80435  OF WHICH:
              25030  NODES "LAGRANGE"
 --- FULL NUMBER OF EQUATIONS:               191245
 --- SIZE OF THE PROFILE MORSE OF TRIANGULAR HIGHER (FORMAT SCR):      3740478
 --- THUS THE SIZE OF THE MATRIX IS:
 --- INTO SYMMETRICAL     NNZ=      3740478
 --- IN NONSYMMETRICAL NNZ=      7289711

 --- FULL NUMBER OF NODES SLAVES:                  5369


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

MOMENT OF CALCULATION:  2.000000000E-02

-------------------------------------------------------------------------------------------------------------------------
|     CONTACT    |   ITERATIONS   |     RESIDUE     |     RESIDUE     |     OPTION     |     CONTACT     |    REACTIONARY GEOM   |
|                |     NEWTON     |     RELATIVE    |     ABSOLUTE    |   ASSEMBLY     |    DISCRETE     |    MAXIMUM     |
|   ITER. GEOM.  |                | RESI_GLOB_RELA | RESI_GLOB_MAXI |                |   ITERATIONS    |                |
-------------------------------------------------------------------------------------------------------------------------
```

We will use the features of `gdb` (or very other *débuguer*) who allow to stop a program after being themselves attached there. It will be necessary for that to know it `PID` achievable Aster which turns in loop. One can for example use the following order (functions only if the achievable one in question is called "`asteru`" and that there is one *job* in the name of the user on the node of calculation):

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

When the job which one wants to auscultate is in parallel, it is delicate to find it `PID` processor `I`. A possibility is to use the tool "`top`", to display the columns `PID` and `PPID` (Relative `PID`) and to go up the number of the process "`asteru`" starting from the number of the temporary repertoire of the form "`proc_pid`" (here `pid` is it `PID` script *Shell* of launching of calculation *Aster*). The idea is that one seeks in the column `PPID`, the number `pid`, one finds then in the column `PID` a new number which one again seeks in the column `PPID`, and so on until arriving at the process "`. /asteru…`".

We carry out then the following line after being ourselves placed in the temporary repertoire of execution:

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

That gives the following thing:

```
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 General the GNU Public License, and you are
welcome to changes it and/or distribute copies of it under certain conditions.
Standard "show copying" to see the conditions.
There is absolutely No warranty for GDB.  Standard "show warranty" for details.
This GDB was configured ace "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… gives.
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 ()
```

We thus stopped the program (it is not more in the state `'running' R` but in the state `'stopped' T`) like the tool shows it *top*.

```
  PID  PPID TO USE      VIRT SWAP  LMBO CODE DATED  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
```

One can from now on make as in one *débuguer*, and to ask where one is to know what occurs:

```
(gdb) where
#0  0x4000000000358a20 in tldlr8_?? unw ()
#1  0x4000000000355ee0 in tldlg3_?? unw ()
#2  0x4000000000357860 in tldlgg_?? unw ()
#3  0x4000000000256b2b0 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 ()
….
```

As a version is used "`nodebug`" one does not have access to the source (and the numbers of line), one would need for that a version compiled with the option "`- G`". Nevertheless one can determine what occurs. Here it is about a calculation of contact which crushes in `tldlr8` the routine which factorizes the complement of Schur of the contact. As there is more than 4000 nodes of active contacts, this factorization is very long (but it is normal).

When one finished in `gdb`, one can be detached from the program then to leave, the execution takes again his course then.

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

1.4 To debug the Python source

When it *bug* relate to the sources Python, it is necessary to use the debugger Python. For that, one still uses `ASTK/To launch "pre"`. After having done something like:

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

One can launch the code under the control of the debugger Python:

```
To start execution in the standard Python débuguer you could:
./asterd /usr/lib/python2.7/pdb.py Python/Execution/E_SUPERV.py - eficas_path \
./Python - orders fort.1.1 - reference mark nun  - num_job 12219 - \ interactive mode
 - rep_outils /opt/aster/outils - rep_mat /opt/aster/NEW9/materiau \
 - rep_dex /opt/aster/NEW9/datg - tpmax 120 - memjeveux 16.0
```

For more details, to see for example: http://docs.python.org/library/pdb.html

1.5 To configure the debugger

The command line used to carry out the debugger (in interactive mode or *post-mortem*) is defined in the files of configuration of ASTK.
To see the command line used in interactive, to make:

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

To see the command line used in *post-mortem*, to make:

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

In general, these orders are defined in the file of configuration of the waiter.
You can modify this value by writing the command line of your choice in $HOME/.astkrc/prefs.

Caution
LE file with to modify is $HOME/.astkrc_salomemeca_VERSION/prefs if ASTK is resulting from Salomé-Meca.

Example for *gdb*, to copy/stick the line:

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

Example for *nemiver*, to copy/stick the line:

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

Example for *idb*, to copy/stick the line:

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

Codes @E, @C,… are replaced by ASTK at the time of launching:

- @E : name of achievable Code_Aster,
- @a : arguments passed to achievable Code_Aster,
- @C : name of the corefile,
- @D : name of the command file for the debugger (which contains where + quit),
- @d : the text corresponding to the orders placed with the debugger,

2 Valgrind

2.1 Presentation

Valgrind is achievable allowing to detect certain programming errors during the execution of a program. The principle of operation of Valgrind is to overload certain functions systems. This is done through a dynamic library and functions such as *malloc, free*, *memcpy* are thus replaced by equivalents instrumented by Valgrind.

More: http://valgrind.org/docs/manual/quick-start.html

or:

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

2.2 Use

To analyze a calculation with Valgrind, the achievable one *Aster* must be compiled with the symbols of debugging (version "debug" with notching in ASTK).

Example of use (to check the Unix program "ls"):

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

More generally, a good command line Valgrind resemble:

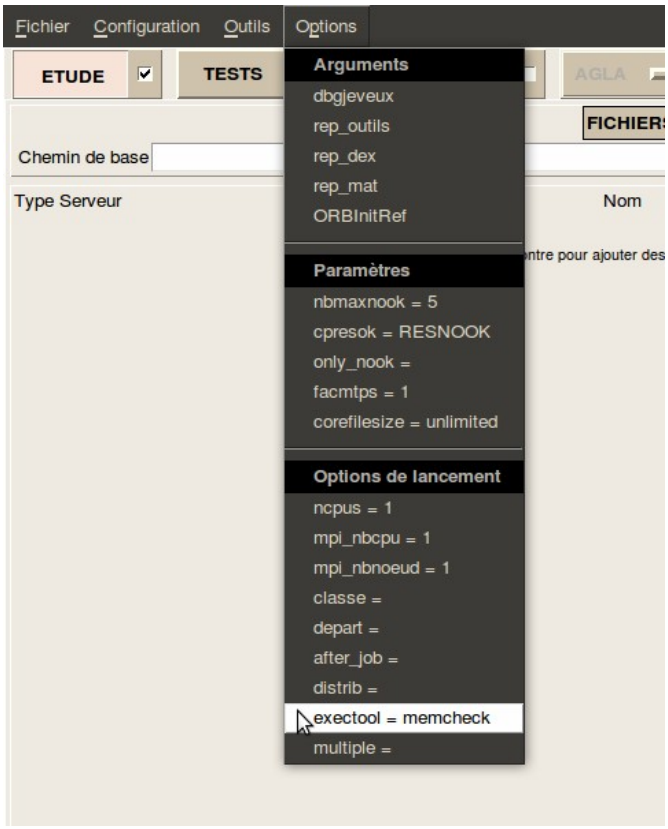
```
valgrind --tool=memcheck --error-limit=no --leak-check=full \
--suppressions=python.supp --track-origins=yes
```

The file python.supp allows to remove the not justified Python errors (Python has its own manager of memory which allows handling not standards). One in general find a specimen of this file in the Linux distributions. On Gauge, this one is in /usr/lib/valgrind/python.supp.

To use Valgrind with *Aster*, it is necessary to be able "to encapsulate" the call to the achievable one. This technique "of encapsulation" can be done in several ways but one details here only simplest (and that which is recommended).
One uses for that the functionality "exectool" of ASTK. One starts by informing in his local file of configuration (located in $HOME/.astkrc/prefs) of the alias worms of the lines of orders which will prefix the line of launching of*Aster:*

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

Then in the menus "Options", one declares exectool=memcheck. Then one launches calculation normally. A message is displayed then to confirm that one wants to launch calculation with the selected tool.



The equivalent with `waf` is obtained while carrying out (to modify it `.export` test to increase the limit in time):

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

Several remarks can be made:

- Execution under `valgrind` can be much longer (30 times sometimes). It is preferable to use achievable "`debug`" so that the diagnosis `valgrind` that is to say more precise (number of line in the sources). To thus think of allocating sufficient time in `ASTK`. It is also sometimes necessary to increase the limit memory under penalty of obtaining a brutal stop in the course of execution without clear information.
- With the option `- - num-callers=n`, the depth is chosen *N* tree of call displayed by `Valgrind`.
- The option `- -track-origins=yes` is available only starting from the versions of Valgrind higher than 3.4.0.

2.3 Decoding

Once launched calculation, the error messages detected by Valgrind will be found then mixed with*output* of*Aster*. They are announced by "`==NumeroDeProcessus==`" and one has 3 types of possible errors in general:

- Use of a not initialized variable
- Reading invalidates apart from a segment report
- Writing invalidates apart from a segment report

Not initialized variable

```
==8906==
==8906== Conditional jump gold move depend one uninitialised been worth (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)
```

In the case of variable initialized, it is possible if the problem does not jump to the eyes to require of `Valgrind` to go up the chain and to indicate in which routine the not initialized variable was created. It is necessary for that to add the option "`- track-origins=yes`". This option is available starting from version 3.4.0.

Reading or invalid writing

```
==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 has 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)
```

This block is presented in two parts. The high part gives the description of the error and its localization in the source. Here in ajellt.f with line 327, one makes a writing of 4 bytes apart from the segment report which had been allocated. For information the line resembled that:

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

The low part gives the origin of the problem. Indeed it is learned that one is located at the address `0x5D3A8E4` with a shift of 0 bytes compared to the segment report in which one is writing (in other words one is at the end of this segment). One thus understands well that if one makes a writing of size 4 bytes, one leaves the segment report. The most invaluable information of the block `Valgrind` is that the object outwards which it is written was allocated in ajellt.f with line 114.

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

By looking at the attributes of the object `LIGRET.LITY`, one realizes that it was dimensioned into hard with a length 1000, from where the problem.

2.4 Errors detected by valgrind but which one can “forget”

It is allowed that errors of the type “ `Conditional jump gold move depend one uninitialised been worth (S)` ” detected on the following routines are not problematic:

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

2.5 Valgrind for the worthless ones

To launch a study with `valgrind`

1) to check that `as_run - showme param memcheck` turn over well a line to carry out valgrind.
2) To multiply the time of the study by 100 in `astk`
3) in `astk/options/exectool` to write `memcheck`
4) to launch the study in `debug`

Analysis of the file .mess

1) to search the occurrences of “ `conditional jump` ”
2) if last routine FORTRAN in the increase is not part of the list of the routines exempted (see §2.4) then there is a true problem: a not initialized variable is declared in this routine. To track this variable `VAR`, one can add `IF (VAR.EQ.XX)` in the source.

3 Debugging JEVEUX

The use of JEVEUX can lead to certain particular errors.

Two tools make it possible the developer to detect these errors:

- procedure in “jeveux debug”
- the routine jxveri.f

3.1 “jeveux debug”

Procedure “jeveux debug” activates itself in ASTK by notching dbgjeveux Options/Parameters/before launching the execution.

The code is carried out then (more slowly) by systematically causing the reading and/or the writing of the objects JEVEUX when they are asked or released from the memory (routines jeveuo, jelibe, jedema). Moreover, when an object jeveux is destroyed (jedetr, jedetc), the zone memory which it occupied is put at “undef”. This behavior of the code makes it possible to cause an error of execution when:

- One continues to use an object after his destruction
- One continues to use an object which “was released”
- One writes in an object whereas one asked for an access in “reading”.

3.2 JXVERI

jxveri.f is the subroutine of *Code_Aster* allowing to detect a crushing in the static storage of JEVEUX.

It is useful when the code stops with one of the following error messages:

```
JEVEUX_15: Crushing upstream…
JEVEUX_16: Crushing downstream…
JEVEUX_17: Broken chaining…
```

The object of debugging is then to locate the instruction which causes the crushing of the memory JEVEUX. For that, one acts by successive iterations.

1st stage
One locates the guilty order while using **BEGINNING (DEBUG=_F (JXVERI=' OUI')**)

2nd stage
One overloads (in debug mode) the routine op00ij corresponding to the guilty order while it “truffant” of **cal jxveri(‘‘,’ ‘)** :

```
subroutine op00ij(...)
        …
        cal jxveri(‘‘,’’)
        block 1
        cal jxveri(‘‘,’’)
        block 2
        cal jxveri(‘‘,’’)
        block 3
        cal jxveri(‘‘,’’)
end
```

During the execution of the code thus overloaded, the code will stop in fatal error with the 1st call to `jxveri` culprit. If for example, it takes place at the end of block 2 (one knows the line thanks to "`traceback`" printed by the debugger "*post-mortem*"), then, one reiterates the process while adding "`cal jxveri`" between the various instructions of block 2. And so on…

In practice, the process converges rather quickly towards the faulty instruction.

4 Other tools

In this part, one describes some tools which also make it possible to find *bugs* or to flush out abnormal behaviors:

- Option "- Checkbounds" compilers
- Comparison of 2 versions different from *Code_Aster*

4.1 Going beyond tables (- CheckBounds)

The compilers have tools making it possible to instrument the code to detect static goings beyond tables (one of the types of *bugs* difficult to find in FORTRAN).

To use these features, one needs recompiler the routines suspect with these options (it is necessary for that to modify it config.txt and to put it in Data in the mitre Overloads) then to carry out the code. If a going beyond occurs, a fatal error with a message will occur.

```
Syntax:
GCC (g77): - fbounds-check
Intel (ifort): - CB
```

Note:

Routines using the COMMON JEVEUX (ZI , ZR ,…) with - CB cannot be compiled because then the execution stops quickly because of the overflow of table ZI (1) .
As a result, the use of - CB is a little complicated: it is necessary to juggle with 2 files config.txt and to preserve the files .o .
Interest of - CB is not enormous because this mechanism does not detect all crushings of tables. So that crushing is detected, it is necessary that the table is local (thus dimensioned in "hard"), or although it is a table argument declared with its exact length (and name not (*)). Other interest of - CB is the detection of crushings of the character strings because in FORTRAN the length of a chain is attached to the chain. Therefore one can make len (chain) on a chain which one received in argument (whereas one cannot make len ()).

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

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

    The current implementation use the “libf2c” library routine “s_rnge” to print the
    diagnosis.

    However, whereas f2c generates has individual check per reference for has multi-
    dimensional array, of the computed offset against the valid offset arranges (0
    through the size of the array), g77 generates has individual check per sub -
    script expression.  This wrestle sum boxes of potential bugs that f2c does not,
    such ace references to below the beginning of year assumed-size array.

    g77 also generates checks for “CHARACTER” substring references, something f2c
    currently does not C.

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

    Note: To provide more detailed information one the offending subscript, g77
    provides the “libg2c” run-time library routine “s_rnge” with somewhat differently-
    formatted information.  Here' s.a sample diagnosis:
    Out Subscript of arranges one spins line 4, procedure rnge.f/bf.
    Attempt to access the 6-HT 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 links (gold statement function) named bf.  The offended
    array is named B.  The offended array dimension is the second for has two-
    dimensional array, and the offending, computed subscript expression was -6.

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

    This indicates that the offended “CHARACTER” variable gold array is named has, the
    offended substring position is the starting (leftmost) position, and the offending
    substring expression is 11.

    (Ideal Though the verbage of “s_rnge” is not for the purpose of the g77 to
    compile, the above adequate information should provide diagnosis abilities to it
    users.)

    Somme of thesis C not work when compiling programs written in FORTRAN:


Intel (ifort): - CB

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

Example of detection of error:

```
forrtl: severe (408): extremely: (2): Subscript #1 of the array RESU
has been worth 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         0000000000499BD 7  aster_oper          2635  astermodule.c
```

4.2 Comparison of 2 versions different from *Code_Aster*

It happens sometimes that two executions different from *Code_Aster* lead to different results.

That can occur:

- With the same version of the code on two different platforms.
- With two different versions (`NR` and `N+1`) on the same platform
- With the same version but with the two achievable ones "`debug`" and "`nodebug`"
- …

The problem to be solved is then to identify the piece of code which has a different behavior for the two executions. To locate the problem, one can start intermediate impressions at some "strategic" places of the code:

- at the time of each call to elementary calculations (routine calcul.f)
- at the time of each call to the routine of resolution of system linear (routine resoud.f)

By doing one `diff` (or one `tkdiff`) on the 2 produced files message, one can locate the place where the 2 versions diverge.

Implementation

To start these impressions, it is necessary to overload the routine calcul.f and/or resoud.f. One modifies the source then by forcing the variable: DBG=.TRUE.

That then involves additional impressions in the file message.

routine `calcul.f`

For example, impressions of the routine `calcul.f` during the calculation of option AMOR_ACOU are:

```
1 &&CALCUL|IN  |PGEOMER | E-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
```

Lines 1,2,3 correspond to the 3 parameters "in" of this option. For each parameter, one prints information on the field associated with this parameter: name of the field, LONMAX of the object containing the values of the field,… and "summarized" (column SOMMR or SOMMI) of the values of the field.

Line 4 indicates that the ligrel on which calculateis calculated contains a grel of elements of the type ACOU_FACE8 and which the routine te00ij.f called is the te0182.f
Line 5 informs about "out" field PMATTTC after elementary calculations of grel ACOU_FACE8 (thus of the te0182.f).

Lines 4 and 5 can be repeated if there is several grel in the ligrel.

Line 6 informs at once "out" after the calculation of all grel.

It can happen that the impressions show that although the fields "in" of an elementary calculation are identical, the fields "out" differ. It is known whereas the problem relates to a precise elementary calculation: OPTION type_element and number of the routine te00ij.f.

Note:

When a field is of whole type, real or complex, the number summarizing the field (SOMMI or SOMMR) is a number obtained in "summoning" the values of the field. Actually, light "a skew" is introduced to allow the detection of a permutation of the values: the vector (1 2 3 4) will lead in general to one SOMMI different from (2 3 1 4).
For the fields of the type CHARACTER , one makes a whole sum (SOMMI) by transforming each character into entirety (function ICHAR).
Caution : a field "in" is almost always different between two executions, it is "the coded" material field ('PMATERC') : it contains addresses JEVEUX who do not have any raison d'être identical. Other objects JEVEUX also almost always have different contents with each execution, they are the objects .TITR who contain the date of the execution in general.
Detail: For each object JEVEUX "summarized", one prints: its name, its "amount" (SOMMI or SOMMR) its LONMAX, its LONUTI, its TYPE (R/C/I/K8/…), a code_retour IRET (if iret/= 0, the object JEVEUX is in a doubtful state) as well as a number IS UNAWARE OF which counts the values "ignored" in the sum (SOMMI or SOMMR). The ignored values are the values 'Not' or invalids (to make the sum): R8MAEM (), R8VIDE (), ISMAEM (),…

routine resoud.f

Impressions of the routine resoud.f are:

```
1 &&RESOUD 2ND MEMBER | &&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
```

Line 1: second member of the linear system
Line 2: values of the eliminated imposed degrees of freedom (char_cine)
Line 3: values of the initial matrix (before factorization)
Line 4: values of the factorized matrix
Line 5: value of the coefficient of conditioning of Lagranges (ddls imposed dualized)
Line 6: values of the solution

If line 1 differs, the problem comes from the manufacturing of the second member of the system.
If only line 4 differs, that translated a problem of factorization (routine preres.f)
If only line 6 differs, the problem comes from the resolution (routine resoud.f).