



## Using a debugger with Abaqus subroutines

Any reasonably large piece of code contains bugs. In scientific codes, bugs can cause an application to crash, hang or produce incorrect answers. The process of finding and resolving bugs in source code is often a tedious task, and can involve hours of work.

### Introduction

One simple and popular approach to debugging is to insert print statements throughout the source code which write out messages or data to the screen when the code is run. This output provides the programmer with information about the values stored in variables, the current position within the code, and so on. However, this approach does not allow the programmer to inspect and control the flow of logic in the program at runtime.

A debugger is a piece of software that typically allows the programmer to pause their program at a certain point in the code. Thereafter, the program can be controlled on a line-by-line basis and the values of variables can be inspected or modified. A debugger is a much more powerful approach to debugging source code than using print statements.

Abaqus is a popular finite element package in the engineering community. Abaqus functionality can be extended through the use of user-defined Fortran subroutines. In this document, the process of attaching a debugger to an Abaqus user-subroutines is described. It is assumed throughout this document that the reader is familiar with debugger usage, such as setting breakpoints, stepping forward through lines of code etc.

## Using a debugger with Abaqus subroutines

Introduction	1
Attaching to an Abaqus subroutine	1
abaqus_v6.env	2
User subroutine	2

### Attaching a debugger to an Abaqus subroutine

Using a debugger with Abaqus subroutines is not as straightforward as with programs for which the entire source is available. Instead of starting Abaqus from inside the debugger, it is necessary to attach the debugger to a running Abaqus process. In this document the Intel debugger is used, but it should be possible to use another debugger by following a similar procedure.

**NOTE:** In the following procedure, Abaqus is run on the login node of the system, which is a shared resource with other users. This is acceptable for small single-core jobs. However, if your debugging run is computationally intensive or needs to run in parallel, the job should be run on the compute nodes of the system.

In the following sequence of steps, the procedure for attaching the Intel debugger to an Abaqus user-subroutine is described.

1. A customised Abaqus environment file (abaqus\_v6.env) needs to be located in the same directory in which the problem is run. This file sets the Abaqus compilation environment so that the user subroutine can be compiled properly for debugging. The file should contain the following, paying particular attention to the indentation of the lines in the file:

```
import os
def prepDebug(var,dbgOption):
    import types
    varOptions = globals().get(var)
    if varOptions:
        # Add debug option
        if type(varOptions) == types.StringType:
            varOptions = varOptions.split()
        varOptions.insert(6,dbgOption)
        # Remove compiler performance options
        if var[:5] == 'comp':
            optOption = ['/O','-O','-xO','-fast','-depend','-vpara']
            for option in varOptions[:]:
                for opt in optOption:
                    if len(option) >= len(opt) and option[:len(opt)] == opt:
                        varOptions.remove(option)
    return varOptions

if os.name == 'nt':
    compile_fortran = prepDebug('compile fortran', '/debug')
    compile_cpp = prepDebug('compile cpp', '/Z7')
    link_sl = prepDebug('link sl', '/DEBUG')
    link_exe = prepDebug('link exe', '/DEBUG')
else:
    compile_fortran = prepDebug('compile fortran', '-g')
    compile_cpp = prepDebug('compile cpp', '-g')

del prepDebug
```

2. Login to Stokes or Stoney on two separate terminals. One of these terminals will be used to start the Abaqus job, the other will be used to run the debugger - this second terminal should have X-forwarding enabled (e.g. login with ssh -X stokes.ichec.ie).

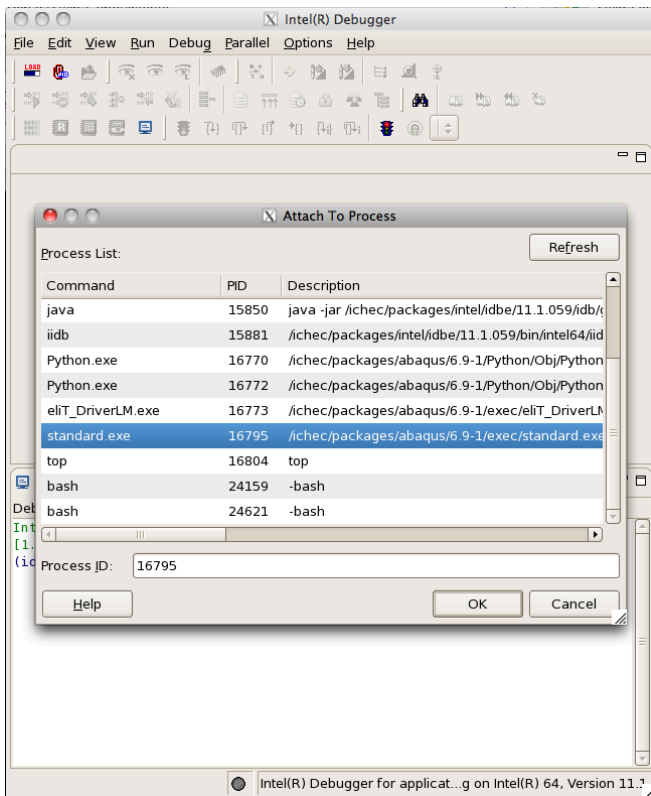
3. The procedure described here requires that the user subroutine be modified temporarily in order to “pause” the program so that the debugger can be attached to the running process. One way to achieve this is to cause the code to enter an infinite loop, e.g.

```
Subroutine umat (stressu,statev,ddsdeu,sse,spd,
+ scd,rpl,ddsddt,drplde,drpldt,stranu,dstranu,
+ time,dtime,temp,dtemp,predef,dpred,cmname,ndiu,
+ nshru,ntensu,nstatv,props,nprops,coords,drot,
+ pnewdt,celent,dfgrd0,dfgrd1,noel,npt,layer,kspt,
+ kstep,kinc)

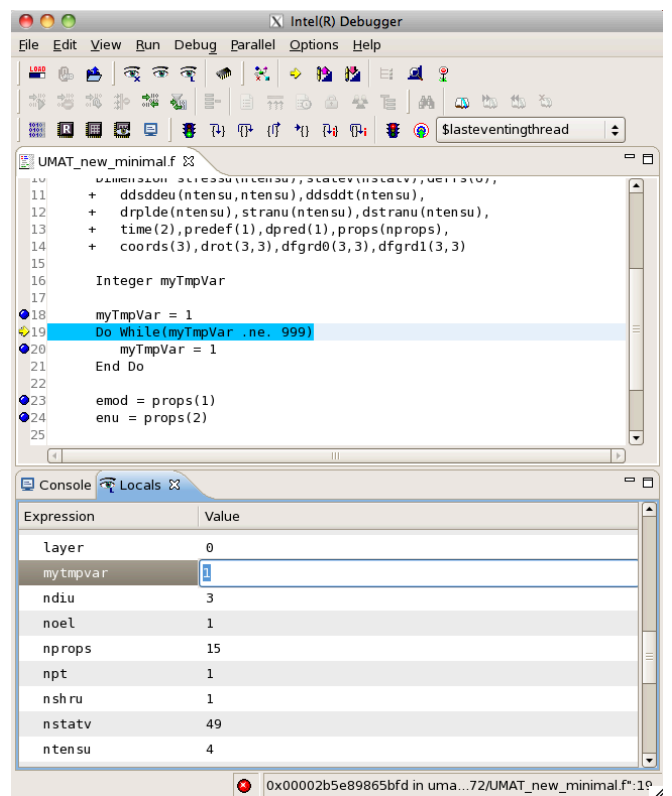
Include 'aba_param.inc'
...
Integer myTmpVar

! The following will create an infinite loop,
! giving sufficient time to attach the debugger to
! the running process once the program is running

Do While(myTmpVar .ne. 999)
    myTmpVar = 1
End Do
```



(a) Attaching the debugger to a running process



(b) Modifying the value of the temporary while-loop variable

Figure 1. Screenshots of the Intel debugger being used to debug an Abaqus subroutine

4. In the terminal with X-forwarding enabled, start the debugger. In this example the Intel debugger is used:

```
user@stokes:> module load intel-db
user@stokes:> idb
```

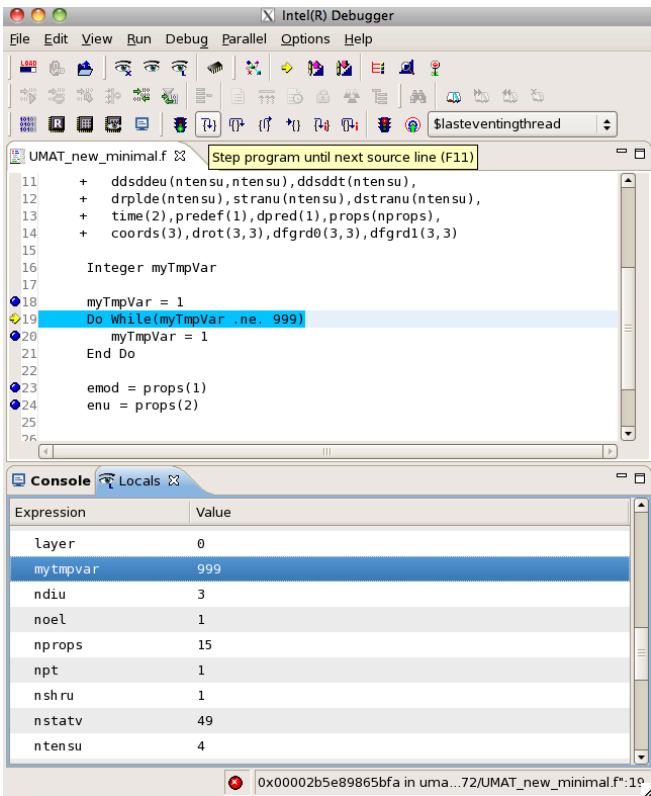
5. In the other terminal, start the Abaqus job as usual, e.g.

```
user@stokes:> abaqus -job myJob -inp myJob.inp -user myUserSub.f int
```

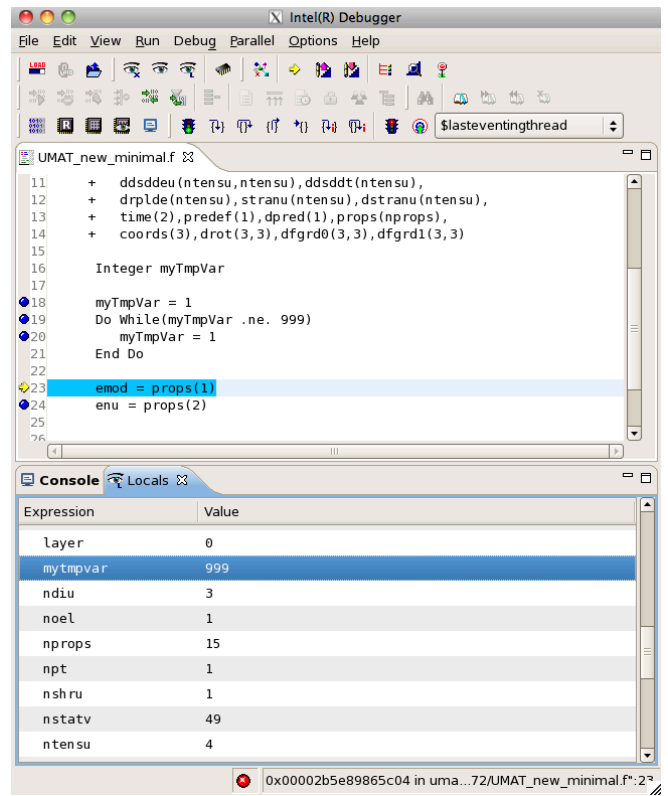
6. At this point go to the debugger, select File->Attach to Process and select the entry containing standard.exe, as shown in Figure 1(a). At this point the source of the user subroutine should appear, with the current execution position (denoted by the yellow arrow) within the loop code inserted previously. In order to break out of this loop it's firstly necessary to step forward though the code until the execution position is at the line containing the Do While statement. Then the value of myTmpVar needs to be changed to 999. As shown in Figure 1(b), to set this value, go to View->Locals, scroll down to myTmpVar and right-click on it and select set value. Stepping forward should cause the code to break out of the loop.

7. As shown in Figure 2(a), at this point the user subroutine should be under the control of the debugger. The code can then be stepped line-by-line, values of variables can be inspected etc.

8. To finish the debugging session, simply close the debugger. Note that you must also ensure that you terminate the Abaqus run in the terminal in which it was started using e.g. Ctrl+C or the kill command



(a) Stepping the source line to exit the temporary infinite loop



(b) The debugger can now be used to debug the code as usual

Figure 2. Screenshots of the Intel debugger being used to debug an Abaqus subroutine