An Introduction to Debugging ADMB Programs

Robert O'Boyle[1]
Beta Scientific Consulting

1 June 2015

This manual is a short description of how to debug ABMB programs using the GDB debugger accessed through the ABMB-IDE. It provides an overview of how to access and run the debugger and the basics of debugging. Only some of the main features of GDB are covered. A full description of the GDB debugger is provided by Stallman et al. (2014). Further, the current version of ADMB has built-in features that also assist in debugging programs, some of which are discussed. The manual does not discuss errors encountered during the translation (from TPL to C++ file) step. These are mostly syntactic and should be self-evident from the messages provided by ADMB. The manual focuses on errors encountered once the ADMB executable file has been built.

The manual benefited from discussion with Chris Grandin and Arni Magnusson which is greatly appreciated.

Confirming the Input data

Before debugging an ADMB program for logic errors (e.g. mis-specified variables, dimensioning issues), it is first necessary to confirm that the input data has been read correctly. This step does not employ the GDB debugger but for completeness, it is discussed here.

The C++ command **`cout`** can be used to print the input data to the ADMB output frame, for example

```
init_int nyrs;
!! cout << "nyrs " << nyrs << endl;
```

Another and perhaps better way to check that the input is being read correctly is to define a C++ macro called **`TRACE`**. This allows output of all designated variables to the output file name of your choice. In the **`DATA_SECTION`**, the command

```
!! TRACE (nobs);
```

outputs the value of **`nobs`** to an output file. The latter is assigned by code in the **`GLOBALS_SECTION`**, specifically

```
#define TRACE(object) tracefile <<"line "<<__LINE__<<", file
"<<__FILE__<<", "
```

1 With contributions by ADMB developers Chris Grandin and Arni Magnusson.

```
<< #object << "\n" << object << "\n" << endl;
ofstream tracefile("input_data_check");
```

The object is the ADMB variable name which is output to **`tracefile`**, the latter assigned to the filename **`input_data_check`** in the same directory as the TPL file. All inputs can be so output and checked.

It is a good idea to insert an **`!! exit (99);`** command at the end of the DATA_SECTION to stop execution of the rest of the ADMB program until the inputs have been confirmed. A number of such statements can be placed throughout the code which are uniquely identified upon execution in the ADMB output frame by a number (99 in this case). These commands are shown in the ADMB manual's Example 10 catch_at_age program (Figure 1).

a) TPL file

b) input_data_check tracefile

Figure 1. Screen shots of catch_at_age model's TPL file (top panel) and input_data_check file produced by the **`TRACE`** macro (bottom panel).

The GDB Debugger

The GNU Debugger (GDB) is a widely used debugger for C and C++ programs in all operating systems and is a good choice as an ADMB code debugger. An Emacs package called the Grand Unified Debugger (GUD) provides an interface to a wide variety of symbolic debuggers. Besides GDB, it supports DBX, SDB, XDB, Perl's debugging mode, the Python debugger PDB, and the Java Debugger JDB.

The purpose of a debugger such as GDB is to allow you to see what is going on "inside" a compiled program while it executes or what another program was doing at the moment it crashed. GDB does four processes (plus others in support of these) to help you debug C++ code:

- Start your program, specifying anything that might affect its behavior
- Make your program stop on specified conditions
- Examine what has happened, when your program stopped
- Change your program, so you can experiment with correcting the effects of one bug and go on to learn about another

These processes will be explored below.

Starting and Stopping the GDB Debugger

To use the GDB debugger, with the TPL file open, click ADMB / Target -g: Debug. This tells ADMB to include debugging symbols in the EXE file, which ensures that the debugging of the CPP file can occur once the executable is created. This step must occur before undertaking any debugging.

Next, compile and build the ADMB program. This produces the EXE file with the GDB debugging symbols included. This step also highlights the need to ensure that no errors encountered in the compile step (or else the EXE file can't be created!).

Now click Tools / Debugger (GDB) and hit enter. The left-hand window now shows the GUD console while the right-hand window shows the CPP file (Fig. 2). The GUD buffer presents information such as GDB version, configuration, copyright and so on. It also ends with the (gdb) prompt, letting you know that GDB is ready for commands. GDB is a command-line driven debugger which can be used as such within the IDE. The most basic GDB commands are linked to the toolbar buttons at the top of the frame. The GUI dropdown menus specific to the debugger (the rest are the same as in ADMB proper) when in the GUD console include Gud, Complete, In/Out and Signals. Additional commands found in the GBD manual can be entered on the (gdb) prompt, which allows the complete functionality of the debugger within the IDE. Any commands entered on the (gdb) prompt can be shortened e.g. r for **`run`**. GBD has six buffers which can be displayed by invoking this command. These come in handy during the debugging. For instance, clicking Gud / GDB-Frames / Stack will allow you to see the information in the stack frame. Each time your program performs a function call, information about the call is generated. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a stack frame. When your program stops, the GDB commands for examining the stack allow you to see all this information.

Figure 2. GUD (left-hand) and CPP file (right-hand) displayed upon invoking the Tools / Debugger (GDB) command. Note (gdb) prompt at bottom of the GUD console.

Up until this point, you haven't actually started to debug the ADMB program (you have only opened the GDB debugger). To start the debugger, you can either click the run icon (see Table 1 below) in the Gud toolbar, click on Gud / run or type **`run`** `or r` in the command line prompt. All have the same effect. The program will run, producing output to the ADMB output frame, but may stop. Repeatedly pushing enter will cause the program to restart from its stop point. When the program finishes, executing run again will start the program from the beginning but as a new thread, indicated by the GDB statement

(gdb) [New Thread 2600.0xd78]

Threads of a single program are akin to multiple processes - except that they share one address space (that is, they can all examine and modify the same variables). On the other hand, each thread has its own registers and execution stack, and perhaps private memory. This manual won't be getting into threads, which are explained in section 4.1 of the GDB manual.

To quit the GDB debugger, either type quit or q or click Signals / EOF or Signals / QUIT. GDB can be restarted by once again selecting Tools / Debugger (GDB).

The Debugging Process

The general idea in debugging is to first identify the section of code that has an error, then examine the program's behaviour to identify the specific error and finally to correct the error. The first step is accomplished through the use of 'breakpoints' which instruct the program to start and stop at lines that you specify. The second step is accomplished through the use of GDB commands that step through the program line by line. The last step involves switching between the TPL and GDB buffers in the IDE.

Breakpoints

When an error is encountered while running an ADMB executable, the message sent to the ADMB output buffer can be something like

```
Incompatible bounds in dvar_vector& dvar_vector::operator =
(const dvar_vector& t)

Process Catage_with_gdb exited abnormally with code 21
```

which provides little information on where the error might be. A way to find out which part of the code has the offending error (at least the first one) is to set a 'breakpoint' just before the error and then run the program, stepping through each line of code to see if the program is operating as intended. A breakpoint is set by clicking in column 1 of the desired line of the C++ code. A red dot appears at the beginning of the line. One can also establish breakpoints by typing **`break n`** where n indicates the desired line number of the C++ code. Clicking on the red dot removes the breakpoint while typing **`disable n`**, where n is the breakpoint number, disables it. Typing **`enable 1`** re-activates breakpoint number 1. Typing **`disable`** disables all the breakpoints while **`enable`** re-establishes these.

But how to find the line with the error? ADMB provides a useful tool in this regard. During the TPL to CPP translation stage, a function called ad_boundf is added. You can find it at the very bottom of the CPP file (Figure 3). Setting a breakpoint on the line with exit(i) and then running the program will cause it to stop at the line with the offending error. The stack frame (mentioned above) includes information on the line with the error. Typing backtrace or bt outputs the stack frame when program stopped. Each line of the stack frame indicates the calls used to get to that line of the code - #0 being ad_boundf, #1 being the call before it and so on down to the end of the stack. About the 3rd or 4th line in the stack indicates the line of code which has the error. Typing frame 4, for instance, outputs the offending line of code. Alternatively, clicking Gud/GDB-Frames/Stack provides the same information but allows you to click on the appropriate stack line which brings you to the line of code with the error (Figure 4).

Figure 3. Ad_boundf function at bottom of CPP file; note that the exit(i) line has a breakpoint set

Figure 4. Stack frame window produced by clicking Gud/Gdb_Frames/Stack; line 4 indicates the code line with the error which when clicked brings the window to that line of code.

Now, set a breakpoint just above the offending line of code and type run or r to rerun the program. The program will stop just above the error which allows you to step through this part of the code to find the source of the error (see below).

As seen, breakpoints are essential to debugging. For each breakpoint, you can add conditions to control in finer detail whether your program stops (see details in the GDB manual). The use of watchpoints and catchpoints are also described in the GDB manual. A watchpoint is a special breakpoint that stops your program when the value of an expression changes, while a catchpoint is another special breakpoint that stops your program when a certain kind of event occurs.

Of course, if you have a good idea where the error is, you can simply set a breakpoint above the suspected code and examine this code in detail using the commands in the next section.

Stepping through the Code

The basic commands to examine C++ code using the GDB debugger are summarized in Table 1. To see how these work, it is best to experiment with some ADMB code.

Typing **`next`** or clicking on the next icon executes the next line of code. Typing **`step`** or clicking on the step icon will enter a function while typing **`finish`** or clicking on its icon steps out of the function.

Typing **`print var`** outputs the value of var to the Gud console. ADMB variables cannot be displayed with this command. For these, it is necessary to use **`print pad(var)`**. As of the current version of the debugger, not all ADMB variables can be displayed with the **`pad()`** command. In these cases, insert a **`cout`** command in the TPL file, which unfortunately implies recompilation and building of the program.

If the program freezes for some reason, you can click Signals / EOF to exit the debugger and then click Tools / Debugger (GDB) to start it again. Then place a breakpoint close to the point where the freeze occurred, click run and start debugging again.

Table 1. Basic GDB commands used when inspecting C++ code. GDB commands can be shortened, often to the first letter.

GDB Command	IDE ICON	Purpose
run		Run or execute the program
continue		Continue or go to next breakpoint
next		Advance execution to next line of code; typing n 4 advances execution 4 lines
step		Step into function or subroutine
finish		Finish or step out of function
print var	none	Print value of var; this only works with integers

print pad(var)	none	Print value of ADMB variable var; pad is short for print AD
cout << "var "<< var << endl;	none	Inserted in TPL file; Print value of ADMB variable when print pad(var) does not work

Correcting Errors

Once an error is found, GDB has tools to correct these in the C++ file, but not the TPL file. Thus, the TPL file should be opened in a frame using Buffers / filename.tpl and the correction made to the file. One can then return to the GDB frame to continue debugging or if preferred, recompile the TPL file with the debugger turned on, open the debugger and continue debugging. For this reason, it is a good idea to have at least the TPL, C++ and GDB frames displayed in the IDE.

Some Examples

The 'catage' example of the ADMB manual will be used to illustrate how to debug ADMB programs.

Error 1: Wrong Indexing

Let's say a loop is indexed incorrectly - maximum of **`nyrs`** rather than **`nages`**. Using the ad_boundf function described above, it was determined that the program stopped in the **`get_mortality_and_survival_rates`** function at line log_sel(j), so a new breakpoint was inserted in the C++ file inside the first loop of that function. . The next command is used to work through the loop, ultimately identifying the source of the array bounds error. The error is corrected in the TPL file, the program recompiled and debugging continued.

Error 2: Wrong Dimension

Let's say that the Z matrix has been incorrectly dimensioned - – column maximum of **`nyrs`** rather than **`nages`**. Using the ad_boundf function, it was determined that the program stopped in the **`get_mortality_and_survival_rates`** function so a new breakpoint was inserted in the C++ file just above the line with the error. The next command is used to work through the code, ultimately producing the bounds error. Without checkpoints (only available in Linux), it is not possible to go back in the code to re-execute the lines. Thus, a next run was conducted and this time, the Z and F arrays printed using p pad(Z or F) before the error which indicate the dimension error. The error is corrected in the TPL file, the program recompiled and rebuilt, and debugging continued.

Final Observations

This manual provides only a very brief introduction to some of the features of the GDB debugger. Debugging with tools such as watchpoints, stacks, threads and so on is left for a future version of the manual. With practice, the lessons learnt using the basic tools discussed here can be significantly expanded upon through reference to the GDB manual.

References

Stallman, R., R. Pesch, S. Shebs, et al. 2014. Debugging with GDB. Tenth edition for GDB version 7.8.1. Free Software Foundation. 740 pp.