ADMB Debugging Tutorial

September 1, 2020
Johnoel Ancheta

Describe how to use a debugger on ADMB programs.

Introduction

Debugging is a method used to help locate errors in ADMB programs. The kind of errors include segmentation faults, division by zero and indexing out of array bounds. A debugger is the application used for debugging. It runs the programs with an interactive shell. The shell can run the program machine instructions to the source code line. This helps find where in the source code that caused the issue. It is able to step through the program code line by line and view values of variables. This tutorial will show how to use MinGW-w64 debugger with ADMB programs in Windows.

GDB Debugger

The GNU Project developed **gdb** for debugging executables. The debugger is a tool for examining the implementation of the source code by running each line one at a time. It can also print the value of instances.

Mapping TPL to C++

To build an executable from a TPL file, it must be first converted to C++ code. The admb script will do the conversion by using a parser. The parser reads the TPL, then generates C++ code from a set of rules. Next the admb script will call the C++ compiler to build the executable from the C++ code. By default, the admb script uses compiler options optimized for speed. It is recommended to use the debugging option with the admb script command to build an executable for debug symbols. The section below will show how to build for debugging.

Mappings

The numbered list below is the sequence of an ADMB program. Each of the SECTIONS are mapped to a C++ function. The C++ functions are used in the debugger to set breakpoints.

1. TOP_OF_MAIN_SECTION

```
int main(int argc,char * argv[])
```

2. PRELIMINARY_CALCS_SECTION

   ```
   void model_parameters::preliminary_calculations()
   ```

3. DATA_SECTION

   ```
   model_data::model_data(int argc,char * argv[])
   ```

4. PARAMETER_SECTION

   ```
   model_parameters::model_parameters(int size, int argc,char * argv[])
   ```

5. PROCEDURE_SECTION

   ```
   void model_parameters::userfunction()
   ```

6. REPORT_SECTION

   ```
   void model_parameters::report(const dvector& gradients)
   ```

7. FINAL_SECTION

   ```
   void model_parameters::final_calcs()
   ```

Debug Release

The ADMB debug release is mainly used for troubleshooting code. It has additional checks and assert statements to ensure valid values. Also debug enables floating point exceptions for overflows, division by zero and invalid function parameters. The debug symbols that are used by a debugger contain source code information. The symbols result in larger size libraries in the distributions. Using the debug release will cause the program to run slower because of the extra testing. It is recommended to use the debug release for debugging only and not in production programs.

Simple Debug

The procedure below will use the debugger gdb to run the simple example. Each step will show and describe gdb shell commands used to control execution of program code. Also, some of the steps will show how to display variables for debugging output.

Below is the simple.tpl file that will be used for the debug session.

```
// Author: David Fournier
// Copyright (c) 2009-2018 ADMB Foundation

DATA_SECTION
  init_int n
  init_vector x(1,n)
  init_vector y(1,n)
PARAMETER_SECTION
  init_number b0
  init_number b1
  vector yhat(1,n)
  objective_function_value f
PROCEDURE_SECTION
  yhat=b0+b1*x;
  f=regression(y,yhat);
```

Prerequisites

- Rtools installed with gdb debugger.

  Rtools 3.5 installations will already come with the gdb debugger.

  For Rtools 4.0 installations, gdb will need to be manually installed. Click Rtools Bash located In the Start Menu -> Rtools 4.0 folder. In the Rtools Bash command windows, type the command below to install gdb. Read the package information link for details https://packages.msys2.org/package/mingw-w64-x86_64-gdb.

```
$ pacman -S mingw-w64-x86_64-gdb
resolving dependencies...
looking for conflicting packages...

Packages (4) mingw-w64-x86_64-expat-2.2.9-9002
             mingw-w64-x86_64-readline-8.0.001-2
             mingw-w64-x86_64-termcap-1.3.1-9002  mingw-w64-x86_64-gdb-9.1-9000

Total Download Size:    3.84 MiB
Total Installed Size:  13.54 MiB

:: Proceed with installation? [Y/n] y
:: Retrieving packages...
 mingw-w64-x86_64...   133.5 KiB   692 KiB/s 00:00 [#####################] 100%
 mingw-w64-x86_64...     5.7 KiB  0.00   B/s 00:00 [#####################] 100%
 mingw-w64-x86_64...   283.7 KiB  2026 KiB/s 00:00 [#####################] 10
0%
 mingw-w64-x86_64...     3.4 MiB  5.44 MiB/s 00:01 [#####################] 100%
(4/4) checking keys in keyring                     [#####################] 100%
(4/4) checking package integrity                   [#####################] 100%
(4/4) loading package files                        [#####################] 100%
(4/4) checking for file conflicts                  [#####################] 100%
(4/4) checking available disk space                [#####################] 100%
:: Processing package changes...
(1/4) installing mingw-w64-x86_64-expat            [#####################] 100%
(2/4) installing mingw-w64-x86_64-termcap          [#####################] 100%
(3/4) installing mingw-w64-x86_64-readline         [#####################] 100%
(4/4) installing mingw-w64-x86_64-gdb              [#####################] 100%
```

- ADMB installed with debug libraries
  - Download ADMB windows release with debug symbols from https://github.com/admb-project/admb/releases/tag/admb-12.2
  - Build ADMB with debug symbols. Read the procedure http://www.admb-project.org/downloads/admb-12.2/BuildingSourceUnix.html. In step two, build with debug symbols with command.

    ```
    admb-12.2-src $ make DEBUG=yes
    ```

Steps

1. Click **Windows Command Prompt** from Windows Start -> Windows System

2. Add paths for ADMB and C++ compiler to the system PATH.

   ```
   C:\> set PATH=C:\ADMB-12.2\bin;C:\Rtools35\mingw_64\bin;%PATH%
   ```

   Note — For Rtools 4, Use C:\Rtools40\mingw64\bin.

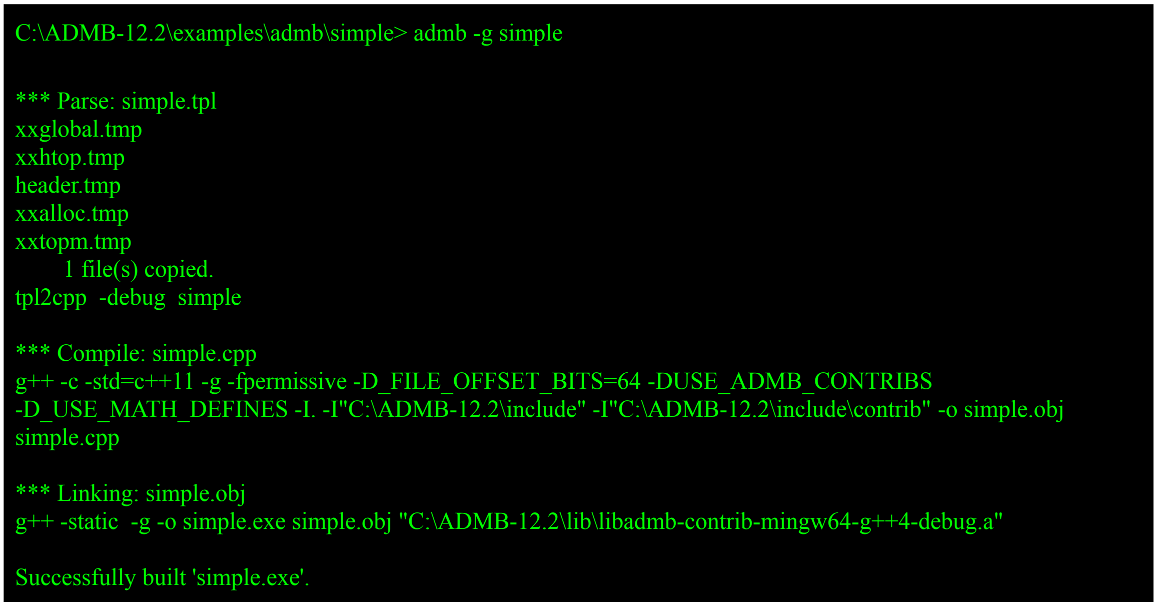
3. Define DEBUG macro to enable additional checks and set compiler CXX macro to g++.

   ```
   C:\> set CXXFLAGS=-DDEBUG
   C:\> set CXX=g++
   ```

4. Change to the simple example directory.

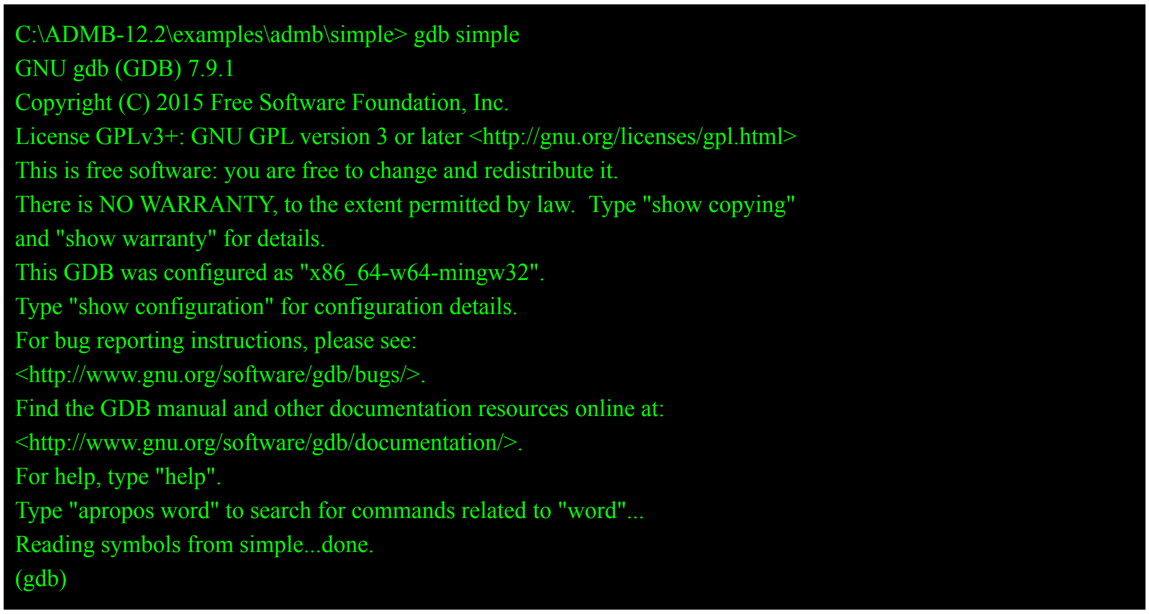
   ```
   C:\> cd \ADMB-12.2\examples\admb\simple
   ```

5. Build the simple example with debug option **-g**.



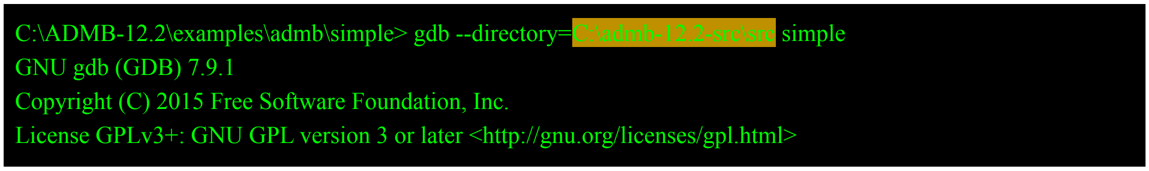
```
C:\ADMB-12.2\examples\admb\simple> admb -g simple


*** Parse: simple.tpl
xxglobal.tmp
xxhtop.tmp
header.tmp
xxalloc.tmp
xxtopm.tmp
        1 file(s) copied.
tpl2cpp  -debug  simple

*** Compile: simple.cpp
g++ -c -std=c++11 -g -fpermissive -D_FILE_OFFSET_BITS=64 -DUSE_ADMB_CONTRIBS
-D_USE_MATH_DEFINES -I. -I"C:\ADMB-12.2\include" -I"C:\ADMB-12.2\include\contrib" -o simple.obj
simple.cpp

*** Linking: simple.obj
g++ -static  -g -o simple.exe simple.obj "C:\ADMB-12.2\lib\libadmb-contrib-mingw64-g++4-debug.a"

Successfully built 'simple.exe'.
```

6. Run program with debugger



```
C:\ADMB-12.2\examples\admb\simple> gdb simple
GNU gdb (GDB) 7.9.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-w64-mingw32".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from simple...done.
(gdb)
```

Set the source directory (if needed). Change the highlighted text directory with the ADMB src directory on the local computer. Ignore warning for recent source files.



```
C:\ADMB-12.2\examples\admb\simple> gdb --directory=C:\admb-12.2-src\src simple
GNU gdb (GDB) 7.9.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

```
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-w64-mingw32".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from simple...done.
(gdb)
```

Alternative - Use **directory** command to set the relocated source directory in gdb shell.

Substitute the highlighted path with the admb src folder from the local machine.

Note — Must use double backslashes ‘\\’ to separate directories.

```
(gdb) directory c:\\Users\\johnoel\\admb-master\\src
```

Ignore the warning message below as long as the source and binary ADMB version are the same.

```
warning: Source file is more recent than executable.
```

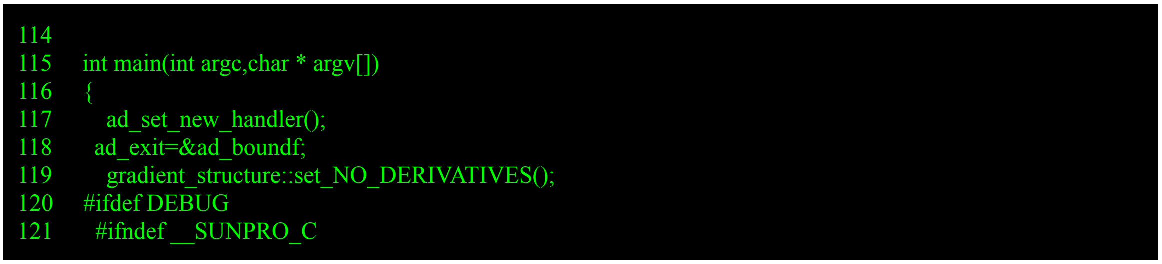
Read the link below for more details.
https://sourceware.org/gdb/current/onlinedocs/gdb/Source-Path.html#set-substitute_002dpath

7. In the gdb shell command, use the **break** command to set a breakpoint at main which is the TOP_OF_MAIN_SECTION. A breakpoint will pause the run of the program before executing code at the specified line.

```
(gdb) break main
Breakpoint 1 at 0x402cfc: file simple.cpp, line 117.
```

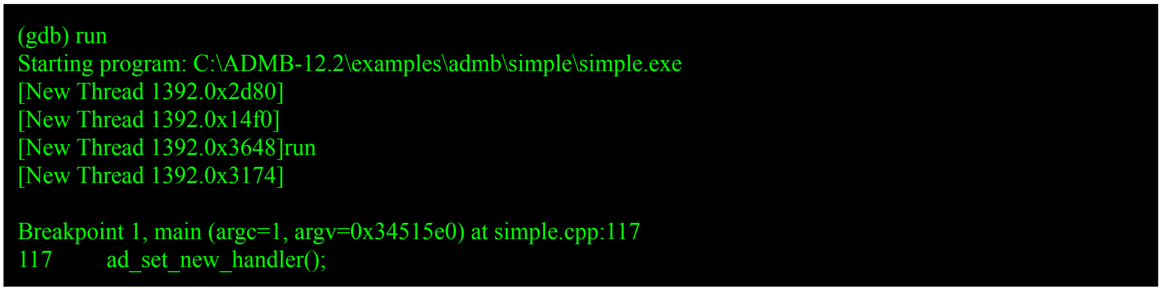
8. Use the **list** command to show the lines of code around line 117 in file simple.cpp.

```
(gdb) list simple.cpp:117
112
113       long int arrmblsize=0;
```



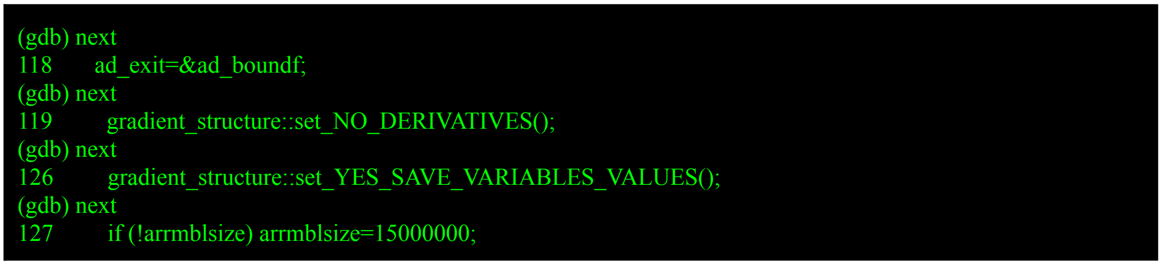
```
114
115    int main(int argc,char * argv[])
116    {
117       ad_set_new_handler();
118     ad_exit=&ad_boundf;
119       gradient_structure::set_NO_DERIVATIVES();
120    #ifdef DEBUG
121     #ifndef __SUNPRO_C
```

9. Use the **run** command to execute the simple program, but will pause at the breakpoint line 117.



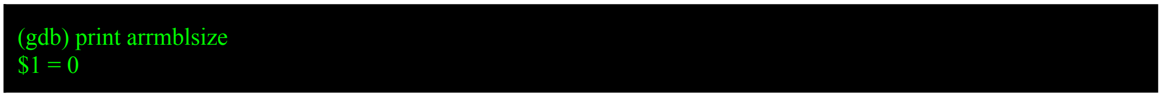
```
(gdb) run
Starting program: C:\ADMB-12.2\examples\admb\simple\simple.exe
[New Thread 1392.0x2d80]
[New Thread 1392.0x14f0]
[New Thread 1392.0x3648]run
[New Thread 1392.0x3174]

Breakpoint 1, main (argc=1, argv=0x34515e0) at simple.cpp:117
117        ad_set_new_handler();
```

10. Use the **next** command to execute the current line, then pause run at the next line.



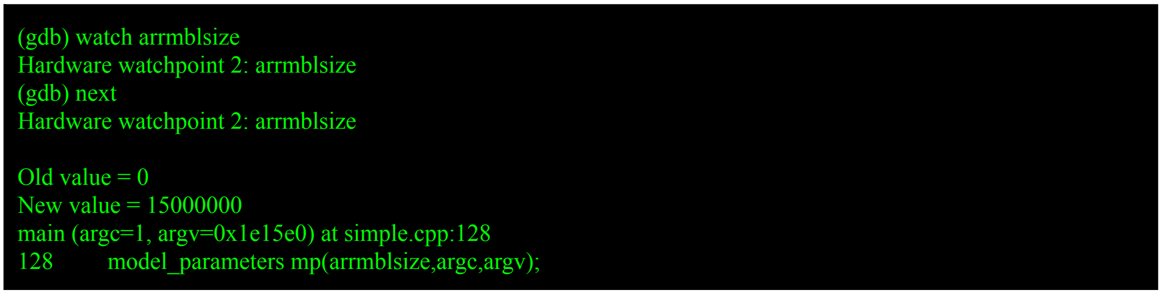
```
(gdb) next
118      ad_exit=&ad_boundf;
(gdb) next
119        gradient_structure::set_NO_DERIVATIVES();
(gdb) next
126        gradient_structure::set_YES_SAVE_VARIABLES_VALUES();
(gdb) next
127        if (!arrmblsize) arrmblsize=15000000;
```

11. Use the **print** command to display the current value.



```
(gdb) print arrmblsize
$1 = 0
```

12. Use **watch** command to notify when the value for arrmblsize in line 113 has changed, then use the **next** command which will execute line 127 (see previous step).

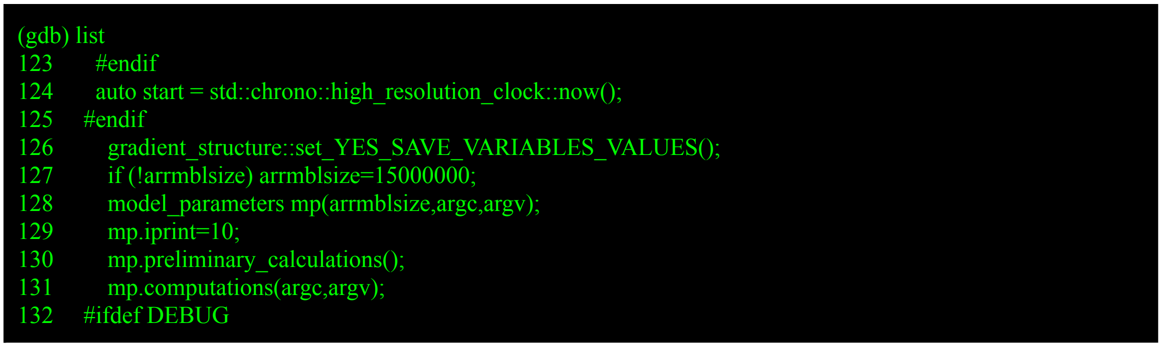


```
(gdb) watch arrmblsize
Hardware watchpoint 2: arrmblsize
(gdb) next
Hardware watchpoint 2: arrmblsize

Old value = 0
New value = 15000000
main (argc=1, argv=0x1e15e0) at simple.cpp:128
128        model_parameters mp(arrmblsize,argc,argv);
```

13. Use **list** command to display lines around the current line. Below are the maps of line numbers for C++ to the TPL sections.

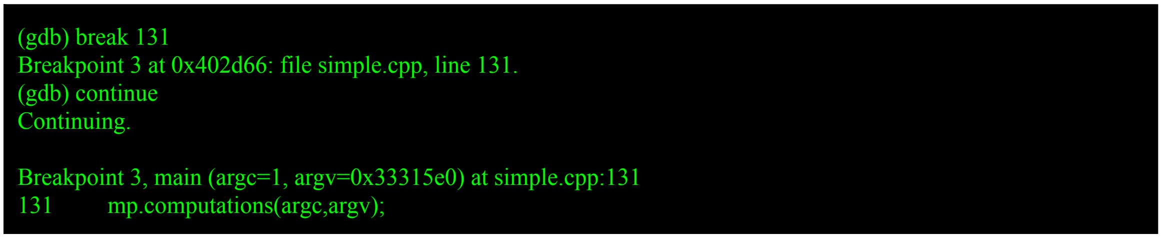
Maps

- Line 128 is the code for the DATA_SECTION and PARAMETER_SECTION.
- Line 130 is the function for PRELIMINARY_CALCS_SECTION.
- Line 131 is the function for PROCEDURE_SECTION, REPORT_SECTION and FINAL_SECTION.



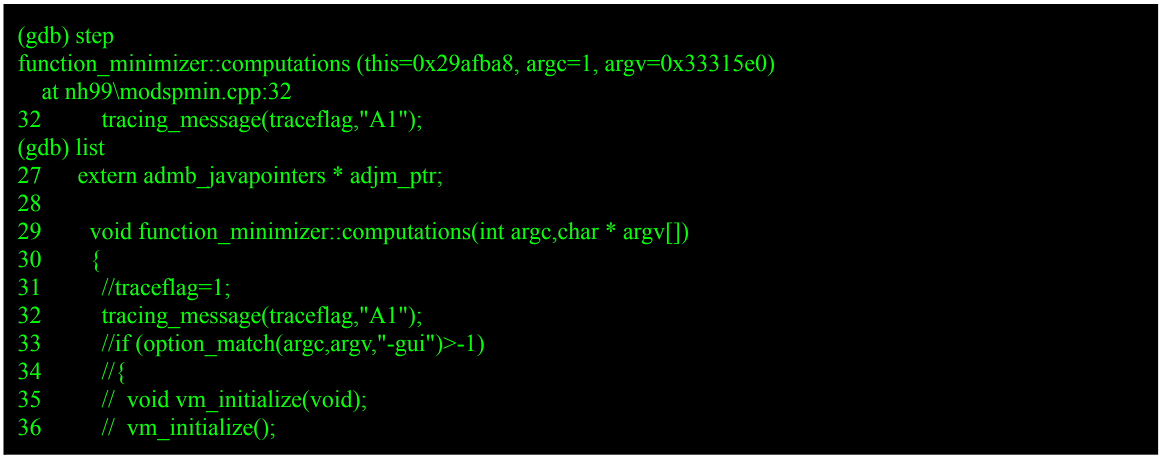
```
(gdb) list
123	    #endif
124	    auto start = std::chrono::high_resolution_clock::now();
125	  #endif
126	      gradient_structure::set_YES_SAVE_VARIABLES_VALUES();
127	      if (!arrmblsize) arrmblsize=15000000;
128	      model_parameters mp(arrmblsize,argc,argv);
129	      mp.iprint=10;
130	      mp.preliminary_calculations();
131	      mp.computations(argc,argv);
132	  #ifdef DEBUG
```

14. Use **break** and **continue** commands to execute till line 131.



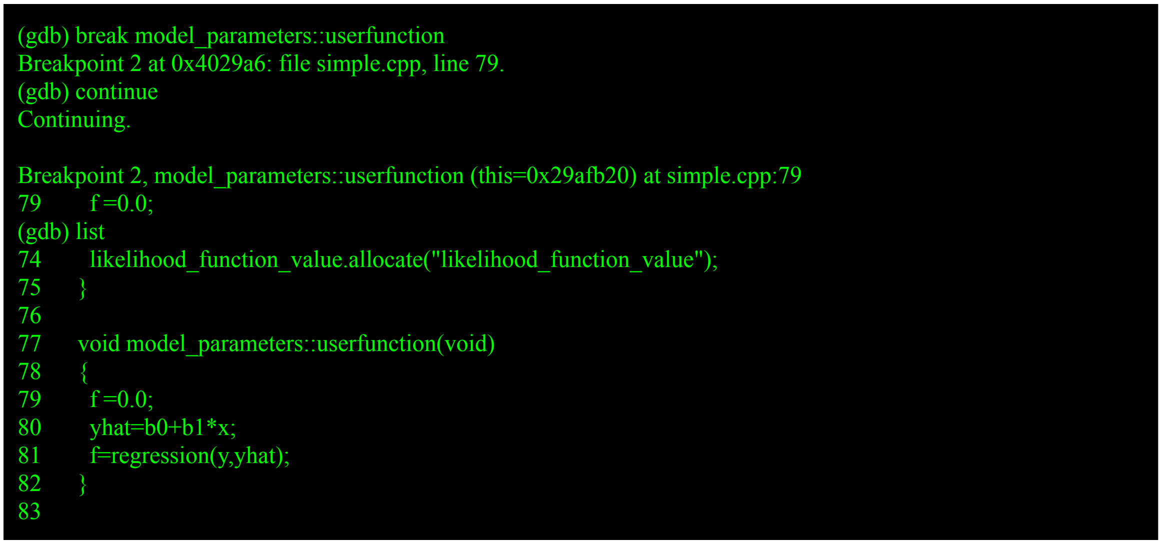
```
(gdb) break 131
Breakpoint 3 at 0x402d66: file simple.cpp, line 131.
(gdb) continue
Continuing.

Breakpoint 3, main (argc=1, argv=0x33315e0) at simple.cpp:131
131	      mp.computations(argc,argv);
```

15. Use the **step** command to execute lines in the mp.computations function, the use **list** to view function code.



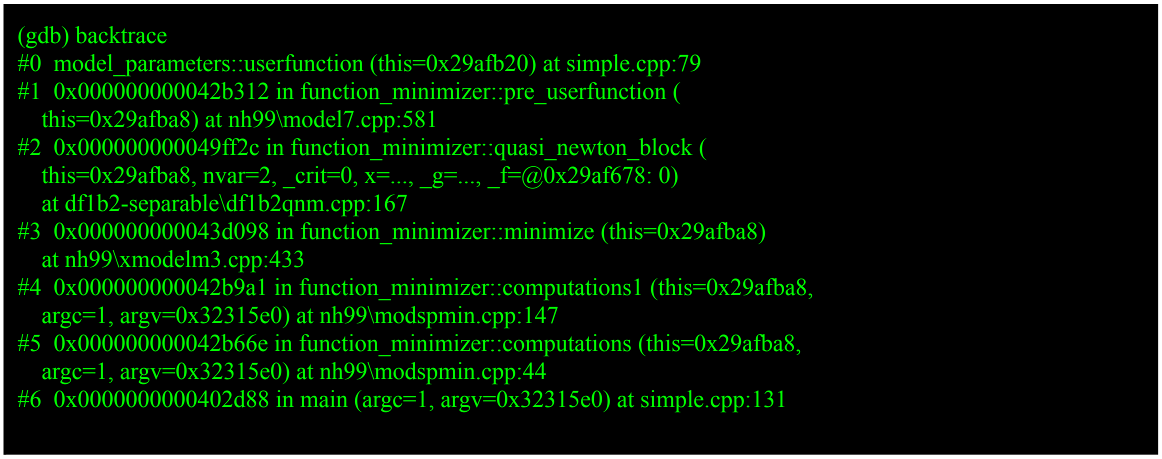
```
(gdb) step
function_minimizer::computations (this=0x29afba8, argc=1, argv=0x33315e0)
    at nh99\modspmin.cpp:32
32	      tracing_message(traceflag,"A1");
(gdb) list
27	  extern admb_javapointers * adjm_ptr;
28
29	    void function_minimizer::computations(int argc,char * argv[])
30	    {
31	      //traceflag=1;
32	      tracing_message(traceflag,"A1");
33	      //if (option_match(argc,argv,"-gui")>-1)
34	      //{
35	      //  void vm_initialize(void);
36	      //  vm_initialize();
```

16. Use the **break** command to stop at the model_paramerrs::userfunction which maps to the PROCEDURE_SECTION in the TPL.



```
(gdb) break model_parameters::userfunction
Breakpoint 2 at 0x4029a6: file simple.cpp, line 79.
(gdb) continue
Continuing.

Breakpoint 2, model_parameters::userfunction (this=0x29afb20) at simple.cpp:79
79        f =0.0;
(gdb) list
74        likelihood_function_value.allocate("likelihood_function_value");
75      }
76
77      void model_parameters::userfunction(void)
78      {
79        f =0.0;
80        yhat=b0+b1*x;
81        f=regression(y,yhat);
82      }
83
```

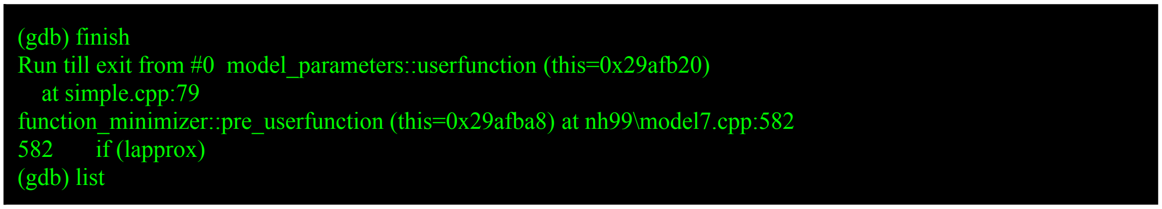
17. Use the **backtrace** command to display the stack of functions.



```
(gdb) backtrace
#0  model_parameters::userfunction (this=0x29afb20) at simple.cpp:79
#1  0x000000000042b312 in function_minimizer::pre_userfunction (
    this=0x29afba8) at nh99\model7.cpp:581
#2  0x000000000049ff2c in function_minimizer::quasi_newton_block (
    this=0x29afba8, nvar=2, _crit=0, x=..., _g=..., _f=@0x29af678: 0)
    at df1b2-separable\df1b2qnm.cpp:167
#3  0x000000000043d098 in function_minimizer::minimize (this=0x29afba8)
    at nh99\xmodelm3.cpp:433
#4  0x000000000042b9a1 in function_minimizer::computations1 (this=0x29afba8,
    argc=1, argv=0x32315e0) at nh99\modspmin.cpp:147
#5  0x000000000042b66e in function_minimizer::computations (this=0x29afba8,
    argc=1, argv=0x32315e0) at nh99\modspmin.cpp:44
#6  0x0000000000402d88 in main (argc=1, argv=0x32315e0) at simple.cpp:131
```

18. Use the **up** and **down** commands to move GDB display to functions in the stack.

19. Use the **finish** command to run the model_parameters::userfunction, then stops at the next line.



```
(gdb) finish
Run till exit from #0  model_parameters::userfunction (this=0x29afb20)
    at simple.cpp:79
function_minimizer::pre_userfunction (this=0x29afba8) at nh99\model7.cpp:582
582        if (lapprox)
(gdb) list
```

```
577          //lapprox->num_separable_calls=0;
578          lapprox->separable_calls_counter=0;
579        }
580      }
581      userfunction();
582      if (lapprox)
583      {
584        if (lapprox->hesstype==2)
585        {
586          lapprox->num_separable_calls=lapprox->separable_calls_counter;
(gdb) backtrace
#0  function_minimizer::pre_userfunction (this=0x29afba8)
    at nh99\model7.cpp:582
#1  0x000000000049ff2c in function_minimizer::quasi_newton_block (
    this=0x29afba8, nvar=2, _crit=0, x=..., _g=..., _f=@0x29af678: 0)
    at df1b2-separable\df1b2qnm.cpp:167
#2  0x000000000043d098 in function_minimizer::minimize (this=0x29afba8)
    at nh99\xmodelm3.cpp:433
#3  0x000000000042b9a1 in function_minimizer::computations1 (this=0x29afba8,
    argc=1, argv=0x32315e0) at nh99\modspmin.cpp:147
#4  0x000000000042b66e in function_minimizer::computations (this=0x29afba8,
    argc=1, argv=0x32315e0) at nh99\modspmin.cpp:44
#5  0x0000000000402d88 in main (argc=1, argv=0x32315e0) at simple.cpp:131
```

20. Use the **continue** command to run program. Since there is a breakpoint at model_parameters::userfunction, the debugger will run the program and stop there.

```
(gdb) continue
Continuing.

Initial statistics: 2 variables; iteration 0; function evaluation 0; phase 1
Function value  2.4980653e+001; maximum gradient component mag -3.6127e+000
Var   Value    Gradient   |Var   Value    Gradient   |Var   Value    Gradient
  1  0.00000 -7.2781e-001 |  2  0.00000 -3.6127e+000 |

Breakpoint 2, model_parameters::userfunction (this=0x29afb20) at simple.cpp:79
79        f =0.0;
(gdb) list
74        likelihood_function_value.allocate("likelihood_function_value");
75      }
76
77      void model_parameters::userfunction(void)
78      {
79        f =0.0;
80        yhat=b0+b1*x;
81        f=regression(y,yhat);
82      }
83
```

21. Use the **print** command to display the value of objective function value f after line 79 and 81 are executed.

```
(gdb) next
80        yhat=b0+b1*x;
```

```
(gdb) print f->v->x
$2 = 0
(gdb) next
81        f=regression(y,yhat);
(gdb) next
82      }
(gdb) print f->v->x
$3 = 24.960634029001518
```

22. Use **delete** command to remove all breakpoints, then use the **continue** command to run the program to the end.

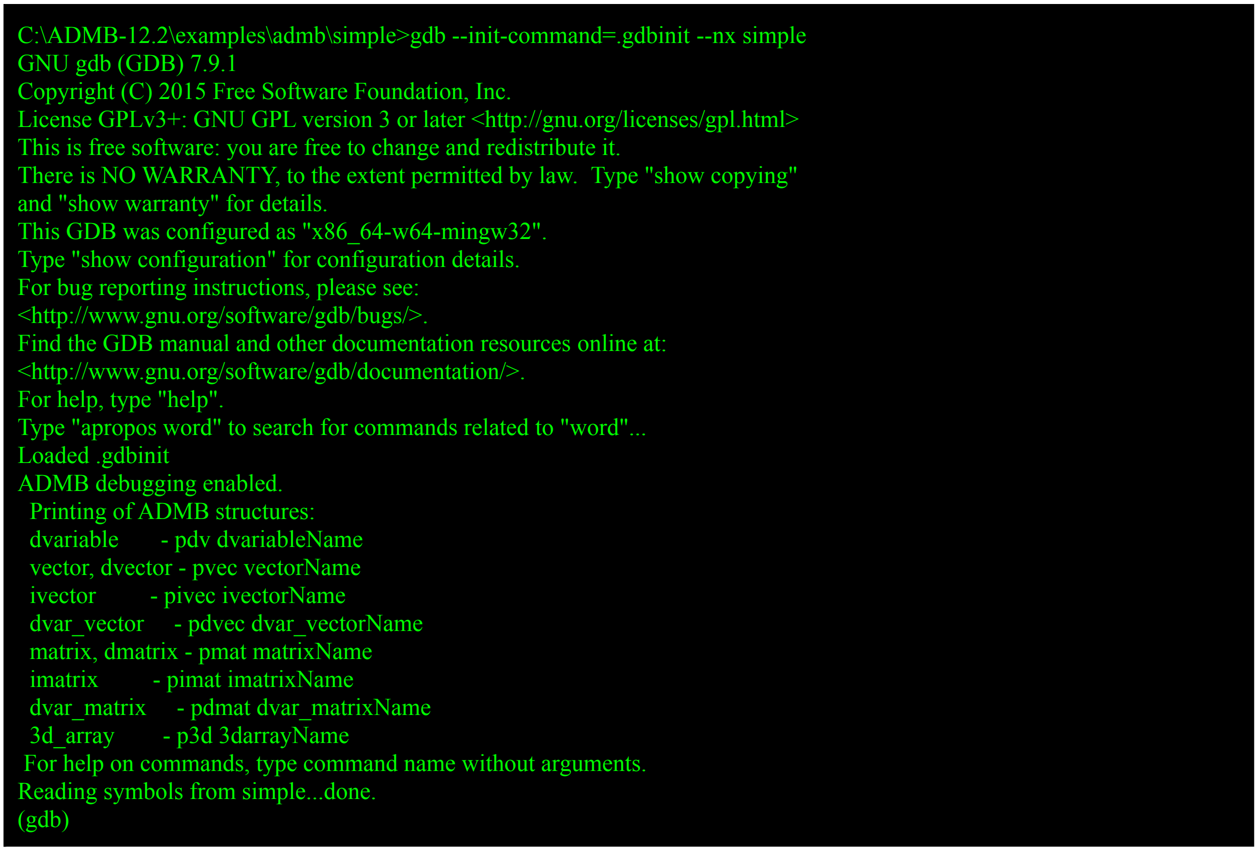
```
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) continue
Continuing.

Initial statistics: 2 variables; iteration 0; function evaluation 0; phase 1
Function value  2.4980653e+001; maximum gradient component mag -3.6127e+000
Var   Value    Gradient   |Var   Value    Gradient   |Var   Value    Gradient
  1  0.00000 -7.2781e-001 |  2  0.00000 -3.6127e+000 |

 - final statistics:
2 variables; iteration 7; function evaluation 19
Function value  3.4513e+000; maximum gradient component mag -7.0014e-005
Exit code = 1;  converg criter  1.0000e-004
Var   Value    Gradient   |Var   Value    Gradient   |Var   Value    Gradient
  1  4.07818 -2.0898e-005 |  2  1.90909 -7.0014e-005 |
Estimating row 1 out of 2 for hessian
Estimating row 2 out of 2 for hessian
[Thread 844.0x30e8 exited with code 0]
[Thread 844.0x1b78 exited with code 0]
[Thread 844.0xd0c exited with code 0]
[Inferior 1 (process 844) exited normally]
```

GDB Scripting

GDB supports scripting in debug sessions which is useful for automating tasks. Below are information for some scripting utilities.

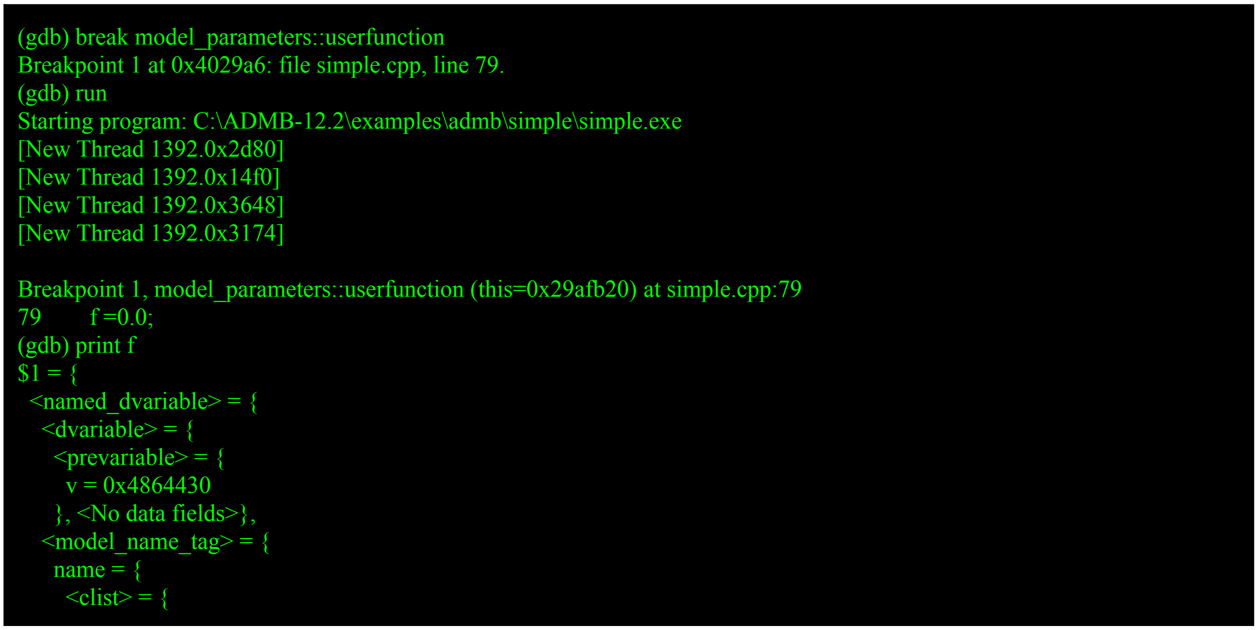
gdbinit

The **print** command does not display the values for some ADMB types. To output the values, the type must be dereferenced like in step 19 above. For higher dimensional arrays, it is more complicated. Chris Grandin (@cgrandin) developed a script for displaying ADMB types in a GDB shell. The script was not packaged with the debug release. So, download the script at https://github.com/admb-project/admb/blob/admb-12.2/utilities/.gdbinit to the simple directory, then use the command below to load the script.



```
C:\ADMB-12.2\examples\admb\simple>gdb --init-command=.gdbinit --nx simple
GNU gdb (GDB) 7.9.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-w64-mingw32".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Loaded .gdbinit
ADMB debugging enabled.
  Printing of ADMB structures:
  dvariable       - pdv dvariableName
  vector, dvector - pvec vectorName
  ivector         - pivec ivectorName
  dvar_vector     - pdvec dvar_vectorName
  matrix, dmatrix - pmat matrixName
  imatrix         - pimat imatrixName
  dvar_matrix     - pdmat dvar_matrixName
  3d_array        - p3d 3darrayName
 For help on commands, type command name without arguments.
Reading symbols from simple...done.
(gdb)
```

The command above shows how to use the display functions. First set a break at the model_parameters::userfunction and **run**, then use the gdb **print** command to display f and yhat in the simple PROCEDURE_SECTION. The resulting output only shows the attributes of the types. The values are not displayed.



```
(gdb) break model_parameters::userfunction
Breakpoint 1 at 0x4029a6: file simple.cpp, line 79.
(gdb) run
Starting program: C:\ADMB-12.2\examples\admb\simple\simple.exe
[New Thread 1392.0x2d80]
[New Thread 1392.0x14f0]
[New Thread 1392.0x3648]
[New Thread 1392.0x3174]

Breakpoint 1, model_parameters::userfunction (this=0x29afb20) at simple.cpp:79
79        f =0.0;
(gdb) print f
$1 = {
  <named_dvariable> = {
    <dvariable> = {
      <prevariable> = {
        v = 0x4864430
      }, <No data fields>},
    <model_name_tag> = {
      name = {
        <clist> = {
```

```
      next = 0x29afdf8
     },
     members of adstring:
     shape = 0x4887be0,
     s = 0x4887c1f ">f"
    }
   }, <No data fields>},
  members of objective_function_value:
  static pobjfun = 0x29afdf0,
  static fun_without_pen = 0,
  static gmax = 0
}
(gdb) print yhat
$2 = {
  <dvar_vector> = {
   va = 0xceea038,
   index_min = 1,
   index_max = 10,
   link_ptr = 0x4887c60,
   shape = 0x4887cc0
  },
  <model_name_tag> = {
   name = {
    <clist> = {
     next = 0x29afd98
    },
    members of adstring:
    shape = 0x4887a60,
    s = 0x4887a9f ";yhat"
   }
  }, <No data fields>}
```

To display the values, use the commands below from the .gdbinit script commands.

```
(gdb) pdv f
pdv: dvariable value = 24.980653
(gdb) pdvec yhat
0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
0.000000
```

GDB scripting is a tool that simplifies code testing. It can script multiple commands to display variables and step through the source code. Below are a few links to show more advanced methods of scripting.

Additional Reads

- https://sdimitro.github.io/post/scripting-gdb/
- https://condor.depaul.edu/glancast/373class/docs/gdb.html

Bug Fix

Below is the bug fix procedure that is used.

Procedure

1. Must be able to duplicate the error. Remote debugging or guessing should be avoided.
2. Locate the line of the code when the error occurs. It will provide a clue to the cause of error. Using a debugger is the best tool for locating the error.
3. Understand why it caused an error. Use gdb **watch** command to check for expected values.
4. Create a unit or TPL test that duplicates the error.
5. Correct the error.
6. Run the test suite to ensure the changes do not break other functions.

For the ADMB Project, there are additional steps for testing with online continuous integration servers. Also, admb uses git issues and branches to test and document changes.

GDB FPE Check

The steps below show how to manually check a function for Floating Point Exceptions (FPE) using GDB shell and FPE Check Utility.

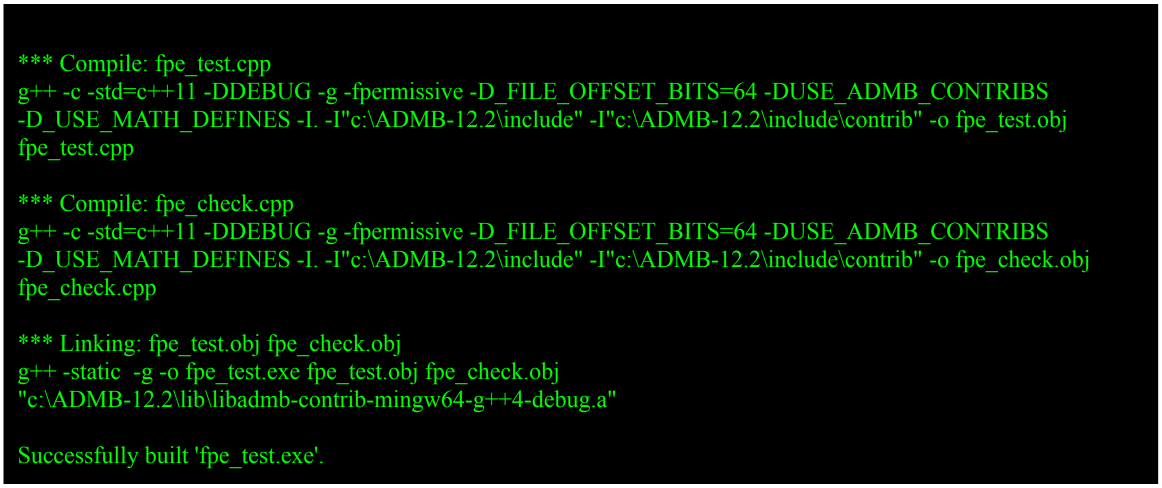
1. Create FPE test directory, the change to that directory.

```
C:\>mkdir fpe-test
C:\>cd fpe-test
```

2. Download FPE test file (fpe_test.tpl) and the FPE Check Utility (fpe_check.cpp) from the ADMB git repository into the directory created in the previous step.

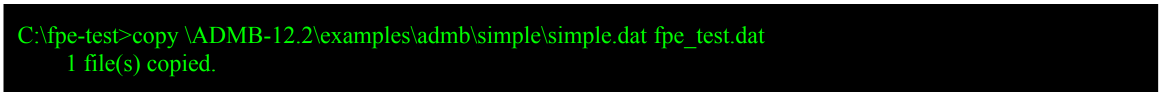
3. Build fpe_test.tpl and fpe_check.cpp with debug flags.

```
C:\fpe-test>admb -g fpe_test.tpl fpe_check.cpp

*** Parse: fpe_test.tpl
xxglobal.tmp
xxhtop.tmp
header.tmp
xxalloc.tmp
xxtopm.tmp
        1 file(s) copied.
tpl2cpp  -debug  fpe_test
```



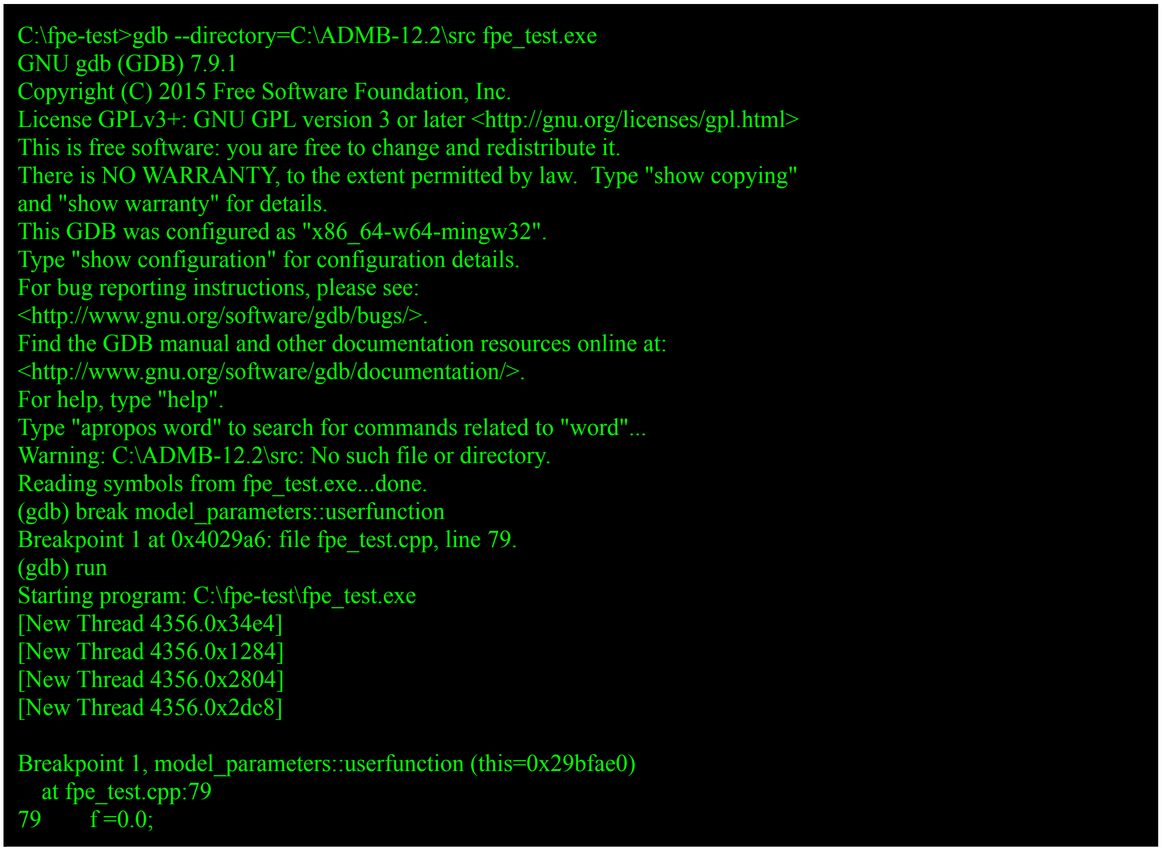
```
*** Compile: fpe_test.cpp
g++ -c -std=c++11 -DDEBUG -g -fpermissive -D_FILE_OFFSET_BITS=64 -DUSE_ADMB_CONTRIBS
-D_USE_MATH_DEFINES -I. -I"c:\ADMB-12.2\include" -I"c:\ADMB-12.2\include\contrib" -o fpe_test.obj
fpe_test.cpp

*** Compile: fpe_check.cpp
g++ -c -std=c++11 -DDEBUG -g -fpermissive -D_FILE_OFFSET_BITS=64 -DUSE_ADMB_CONTRIBS
-D_USE_MATH_DEFINES -I. -I"c:\ADMB-12.2\include" -I"c:\ADMB-12.2\include\contrib" -o fpe_check.obj
fpe_check.cpp

*** Linking: fpe_test.obj fpe_check.obj
g++ -static  -g -o fpe_test.exe fpe_test.obj fpe_check.obj
"c:\ADMB-12.2\lib\libadmb-contrib-mingw64-g++4-debug.a"

Successfully built 'fpe_test.exe'.
```

4. Copy DAT from the simple example into fpt_test.dat.



```
C:\fpe-test>copy \ADMB-12.2\examples\admb\simple\simple.dat fpe_test.dat
        1 file(s) copied.
```

5. Use gdb, then set **breakpoint** model_parameters::userfunction and **run**.



```
C:\fpe-test>gdb --directory=C:\ADMB-12.2\src fpe_test.exe
GNU gdb (GDB) 7.9.1
Copyright (C) 2015 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-w64-mingw32".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Warning: C:\ADMB-12.2\src: No such file or directory.
Reading symbols from fpe_test.exe...done.
(gdb) break model_parameters::userfunction
Breakpoint 1 at 0x4029a6: file fpe_test.cpp, line 79.
(gdb) run
Starting program: C:\fpe-test\fpe_test.exe
[New Thread 4356.0x34e4]
[New Thread 4356.0x1284]
[New Thread 4356.0x2804]
[New Thread 4356.0x2dc8]

Breakpoint 1, model_parameters::userfunction (this=0x29bfae0)
    at fpe_test.cpp:79
79        f =0.0;
```

6. In the beginning of the model_parameters::userfunction, initialize the FPE flags and check that None is detected.

```
(gdb) p fpe_init()
Clear FPE
$14 = void
(gdb) p fpe_check()
Detected FPE:
None
$15 = void
```

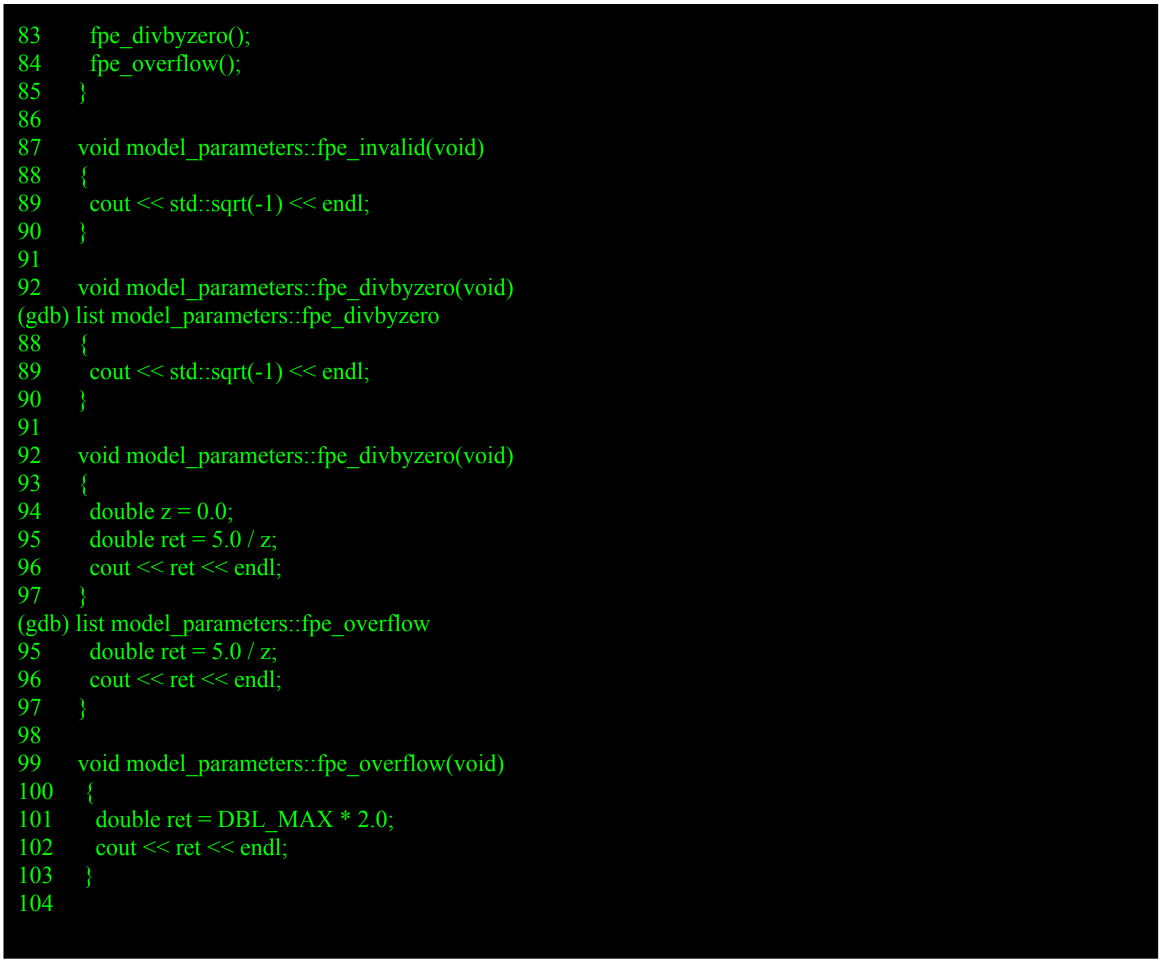
7. Run to the end of the function with the **finish** command, then check FPE flags.

```
(gdb) list 81
76
77      void model_parameters::userfunction(void)
78      {
79        f =0.0;
80        yhat=b0+b1*x;
81        f=regression(y,yhat);
82        fpe_invalid();
83        fpe_divbyzero();
84        fpe_overflow();
85      }
(gdb) finish
Run till exit from #0  model_parameters::userfunction (this=0x29bfae0)
    at fpe_test.cpp:79
nan
inf
inf
function_minimizer::pre_userfunction (this=0x29bfb68) at nh99\model7.cpp:582
582       if (lapprox)
(gdb) p fpe_check()
Detected FPE:

Detected division by zero

Detected invalid argument

Detected overflow
$1 = void
```

8. The beginning of the previous command, the **list** command was used to display the function body around line 81. The functions fpe_invalid(), fpe_divbyzero() and fpe_overflow() have code that would throw FPE exceptions. To view functions use the **list** command with the name of the function.

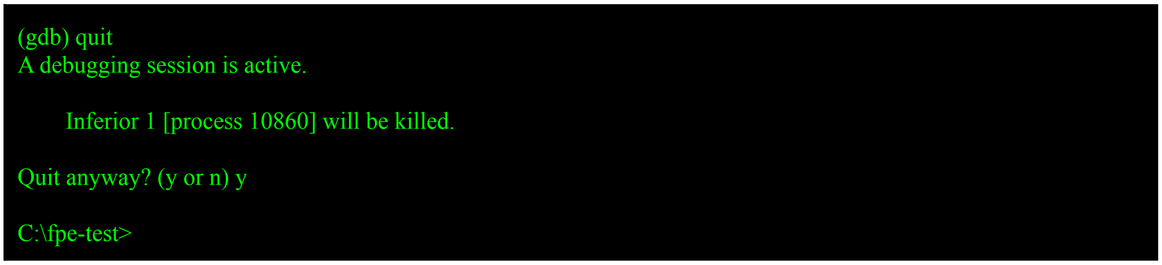
   Note — model_parameters:: is prefixed to the function names.

```
(gdb) list model_parameters::fpe_invalid
```



```
83      fpe_divbyzero();
84      fpe_overflow();
85    }
86
87    void model_parameters::fpe_invalid(void)
88    {
89      cout << std::sqrt(-1) << endl;
90    }
91
92    void model_parameters::fpe_divbyzero(void)
(gdb) list model_parameters::fpe_divbyzero
88    {
89      cout << std::sqrt(-1) << endl;
90    }
91
92    void model_parameters::fpe_divbyzero(void)
93    {
94      double z = 0.0;
95      double ret = 5.0 / z;
96      cout << ret << endl;
97    }
(gdb) list model_parameters::fpe_overflow
95      double ret = 5.0 / z;
96      cout << ret << endl;
97    }
98
99    void model_parameters::fpe_overflow(void)
100    {
101      double ret = DBL_MAX * 2.0;
102      cout << ret << endl;
103    }
104
```

9. Use **quit** command to exit the debugger. Type y to quit when prompted.



```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 10860] will be killed.

Quit anyway? (y or n) y

C:\fpe-test>
```

GDB FPE Check Script

Instead of manually retyping each command to check if a function generated a FPE, a script can simplify it.   The steps below show how to use the FPE Check script to check multiple functions for Floating Point Exceptions (FPE) using GDB shell and FPE Check Utility.

Note — The script and fpe_check.cpp utilities can be used to check for FPE in any ADMB program.

1. Download fpe_check.gdb into the directory create above.

2. Run the command below to run gdb with script.

```
C:\fpe-test>gdb --command=fpe_check.gdb fpe_test
```

   Note — The above command will output a lot of lines.

Ideas

Below are some ideas for development of the next release.

TPL FUNCTION in C++

Be able to define a TPL FUNCTION in a C++ source file instead of the TPL. This will decrease the total line count of existing TPL files. Using multiple files will make developing and code maintenance easier. Also, debuggers can step through the actual C++ function instead of the mapped version. The admb script is able to build and link in multiple C++ files with the TPL.

Workflow

Describe the potential workflow to move an FUNCTION from the catage example into a C++ file.

***Note** — This is only a proposed feature, it has not been implemented into ADMB.*

1. Change to the catage example directory.

```
C:\>cd admb-12.2\examples\admb\catage
```

2. Copy function body of FUNCTION evaluate_the_objective_function from catage.tpl into C++ source file evaluate_the_objective_function.cpp.

   In catage.tpl,

```
...
FUNCTION get_catch_at_age
  C=elem_prod(elem_div(F,Z),elem_prod(1.-S,N));
```

```
FUNCTION evaluate_the_objective_function
 // penalty functions to ``regularize '' the solution
 f+=.01*norm2(log_relpop);
 avg_F=sum(F)/double(size_count(F));
 if (last_phase())
 {
  // a very small penalty on the average fishing mortality
  f+= .001*square(log(avg_F/.2));
 }
 else
 {
  f+= 1000.*square(log(avg_F/.2));
 }
 f+=0.5*double(size_count(C)+size_count(effort_devs))
  * log( sum(elem_div(square(C-obs_catch_at_age),.01+C))
  + 0.1*norm2(effort_devs));

REPORT_SECTION
 report << "Estimated numbers of fish " << endl;
 report << N << endl;
 report << "Estimated n
...
```

To create evaluate_the_objective_function.cpp, modify the template below.

```
// Replace TPL_NAME with the filename of  TPL
#include <TPL_NAME.htp>

//  Replace FUNCTION_NAME with the function from TPL
void model_parameters::FUNCTION_NAME()
{
  // Move function body contents here.
}
```

The evaluate_the_objective_function.cpp should like the text below.

```
#include <catage.htp>

// Moved the body of evaluate_the_objective_function from TPL
// Notice the body below is the same as in the catage.tpl
void model_parameters::evaluate_the_objective_function()
{
 // penalty functions to ``regularize '' the solution
 f+=.01*norm2(log_relpop);
 avg_F=sum(F)/double(size_count(F));
 if (last_phase())
```

```
  {
   // a very small penalty on the average fishing mortality
   f+= .001*square(log(avg_F/.2));
  }
  else
  {
   f+= 1000.*square(log(avg_F/.2));
  }
  f+=0.5*double(size_count(C)+size_count(effort_devs))
   * log( sum(elem_div(square(C-obs_catch_at_age),.01+C))
   + 0.1*norm2(effort_devs));
}
```

3. Delete the function body in the TPL, and declare only the function name in the catage.tpl file with a semicolon(;) at the end. The result will look like the following.

```
…
FUNCTION get_catch_at_age
   C=elem_prod(elem_div(F,Z),elem_prod(1.-S,N));

FUNCTION evaluate_the_objective_function;

REPORT_SECTION
   report << "Estimated numbers of fish " << endl;
   report << N << endl;
   report << "Estimated numbers in catch " << endl;
...
```

Note - The body evaluate_the_objective_function is no longer defined. The code was moved and defined in a C++ file. The declaration is still needed by the admb script to add member function into the model_parameters class.

4. Build catage executable using catage.tpl and evaluate_the_objective_function.cpp using the command below.

```
C:\admb-12.2\examples\admb\catage> admb catage.tpl evaluate_the_objective_function.cpp
```

More Information

Websites

- ADMB Project
- GDB

- MinGW-w64
- Rtools

Mailing Lists

- ADMB Users is the main mailing list. Email users@admb-project.org to contact the group.
- ADMB Developers is for core team discussions. The email is developers@admb-project.org.