

Floating Point Exceptions and Signals in gcc/gfortran

by Milad Fatenejad

Thanks to Jim Porter for figuring out how to make this work with gcc/g++

When first writing any computer program, almost everyone introduces at least a few bugs. Often times, these bugs manifest themselves when you do bad things with your math. For example, you might divide by zero in your code or take the square root of a negative number. It would be nice if your program automatically told you when things like this happen. Ideally, you would also be informed of exactly where these errors appeared. By default gcc, g++ and gfortran do not provide this information, but they be made to provide it.

There are four main types of floating point errors that are of interest. These are described below:

- **Overflow:** This error occurs when a number becomes too large. For example if you take the exponential of a large number, say 1000, the result will be too large to store in a double precision number. Typically, double precision numbers (64 bits) can store values up to roughly 10^{308} .
- **Underflow:** This error occurs when a number gets too close to zero. The smallest positive number that can be represented using double precision is roughly 10^{-308} . These can occur if you try to evaluate the exponential of a very large negative number.
- **Divide by Zero:** These errors occur when you divide a non-zero number by zero
- **Invalid:** This is a catch all for a lot of different errors. For example, when you try to evaluate the square root of a negative number or if you divide zero by zero.

There are times when you may want to ignore some of these errors. For example, consider the following code:

```
var1 = 10.0 + exp(-1000.0);
var2 = 10.0 + 1.0/(exp(1000.0) + 1);
```

The first line generates an underflow error, but we might want to ignore it and just let `exp(-1000.0)` evaluate to 0.0, and let `var1` evaluate to 10.0. The second line above presents a similar situation. If we don't throw an error, `1/(exp(1000)+1)` will evaluate to zero which is probably what we want. However, we generally always want to be informed when divide by zero and invalid errors occur. It is up to you to decide which errors cause your program to abort and which do not.

Handling Floating Point Errors in gfortran

I'll start by discussing how to enable floating point errors using gfortran, because it is more straightforward than using gcc/g++. Consider the following code that I've placed in a file called **signal.f90**:

```

Line
1      subroutine invalid_sqrt(arg)
2          real, intent(in) :: arg
3          real :: b
4
5          b = sqrt(arg)
6          print *, "Square root result = ", b
7      end subroutine invalid_sqrt
8
9      subroutine overflow_exp(arg)
10         real, intent(in) :: arg
11         real :: b
12
13         b = exp(arg)
14         print *, "Overflow results = ", b, 1.0/b
15     end subroutine overflow_exp
```

```

16
17      subroutine divide_by_zero(arg)
18      real, intent(in) :: arg
19      real :: b
20
21      b = 0.0/arg
22      print *, "Divide by zero results = ", b
23      end subroutine divide_by_zero
24
25      subroutine test
26      call invalid_sqrt(-1.0)
27      call overflow_exp(1.0e5)
28      call divide_by_zero(0.0)
29      end subroutine test
30
31      program main
32      call test
33      end program main

```

The example below demonstrates what happens if we compile and run the program:

```

gfortran signal.f90 -o signal
./signal
Square root result =           NaN
Overflow results =      +Infinity  0.0000000
Divide by zero results =           NaN

```

The program runs three tests, in the subroutines `invalid_sqrt`, `overflow_exp`, and `divide_by_zero`, which generate floating point errors. However, we are not currently trapping these errors, so the program runs without giving an indication that anything has gone wrong. Notice, that in the case of the `overflow_exp` test, we might not want our program to be interrupted, but we definitely want to be informed if our code is taking the square root of negative numbers.

To generate useful error information, just compile the code as follows:

```

gfortran -g -ftrap-fpe=invalid,zero -fbacktrace signal.f90 -o signal
./signal

Program received signal 8 (SIGFPE): Floating-point exception.

Backtrace for this error:
+ /lib/libc.so.6 [0x7f11670ac040]
+ function invalid_sqrt (0x4008A9)
  at line 5 of file signal.f90
+ function test (0x400A96)
  at line 27 of file signal.f90
+ /lib/libc.so.6(__libc_start_main+0xe6) [0x7f11670975a6]

```

Notice, that now, the code freaks out when it gets to line 5, because we take the square root of a negative number. Not only does the program abort, but it tells us exactly where the error occurred. This information can be very useful when writing programs. To enable this capability we had to use a few command line options when we compiled our code. I've repeated the compiling command below:

```
gfortran -g -ftrap-fpe=invalid,zero -fbacktrace signal.f90 -o signal
```

The extra options are described below.

- **-g**: This turns on debugging information. Without this option, the back trace above wouldn't include any line numbers.
- **-ftrap-fpe=invalid,zero**: This option tells the compiler to trap the invalid and divide-by-zero floating point errors. When these errors occur, the program will abort. Note, that the compiler will ignore underflow and overflow errors. To trap all floating point errors, use the option **-ftrap-fpe=invalid,zero,underflow,overflow**.
- **-fbacktrace**: This option tells the compiler to print a back trace when an error is encountered. Without this, the program would simply abort when it encountered an error and print a message, such as "Floating point exception". The back trace tells us exactly where the error occurred.

So, by just adding a few extra options, we can tell the code to automatically detect errors and print exactly where they occur. This can really help you identify and fix errors.

Handling Floating Point Errors in gcc/g++

The process for handling errors in g++ is the same as gcc, so I will simply describe how it is done with g++. Unfortunately, there are no simple command line options to do what we want. We have to make some slight modifications to our code to detect errors using g++.

```

Line
1           #include <iostream>
2           #include <cmath>
3
4           #include <csignal>
5           #include <fenv.h>
6           #include <cstdlib>
7
8           void signal_handler(int sig)
9           {
10          std::cerr << "Error: Floating point signal detected.\n";
11          exit(1);
12          }
13
14          void test()
15          {
16          double d1 = sqrt(-1.0); // This should generate a signal
17          }
18
19          int main()
20          {
21          // Turn on floating point exceptions:
22          feenableexcept(FE_DIVBYZERO | FE_INVALID);
23
24          // Set the signal handler:

```

```

25             signal(SIGFPE, signal_handler);
26
27             // Call function which should cause the program to abort:
28             test();
29
30             std::cout << "Program exited normally.\n";
31             return 0;
32         }

```

The key lines are 22 and 25. These lines tell the compiler to:

1. Enable the divide-by-zero and invalid errors
2. Print an error message and exit when an error is encountered.

Line 22 contains a call to the function `feenableexcept` which is declared in the header file `fenv.h`. We tell `feenableexcept` which errors we want to detect. In this case, we have indicated the divide-by-zero and invalid errors. To detect all errors, we can either write:

```
feenableexcept(FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW)
```

or simply

```
feenableexcept(FE_ALL_EXCEPT)
```

Note that the latter activates all floating point exceptions, including a few we haven't discussed here.

With just line 22, the program would still not do anything when a floating point error, like `sqrt(-1)`, was encountered. This is because line 22 simply tells the computer to send a signal when an error is encountered - a signal which is simply ignored, unless we write a function to handle the signal. On line 25, we use the function `signal` (declared in the header file `csignal`), to indicate that the function `signal_handler`, defined on lines 8 through 12, should be called whenever a floating point error is encountered. Notice that that function prints a one line error message and exits.

If we compile and run our program now, the following is outputted:

```
g++ signal.cpp -o signal
./signal
Error: Floating point signal detected.
```

Notice, that while it did tell us that an error was detected, it does not give us the back trace we need to track down where the error occurred. The easiest way to obtain a back trace is to run the program in a debugger. This procedure is shown below using the debugger `gdb`.

```
g++ -g signal.cpp -o signal
gdb signal

GNU gdb 6.8-debian
Copyright (C) 2008 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-linux-gnu"...
(gdb) run
Starting program: /home/fateneja/Desktop/signalling/signal

Program received signal SIGFPE, Arithmetic exception.
0x0000000040098e in test () at signal.cpp:16
16         double d1 = sqrt(-1.0); // This should generate a signal
(gdb) where
```

```
#0 0x00000000040098e in test () at signal.cpp:16
#1 0x0000000004009e3 in main () at signal.cpp:28
(gdb)
```

On the first line above, we compile the program using the `-g` option to enable debugging symbols. Then we start the debugger `gdb`, using the executable `signal`. We run the program in `gdb` using the `run` command, which quickly informs us that there has been an arithmetic exception on line 16 of the file `signal.cpp`. If you then enter the command `where`, `gdb` will give you the full back trace, as in the FORTRAN example.