Roee Distler

COMP 587

Professor Dewey

14 May 2020

**Testing GLM**

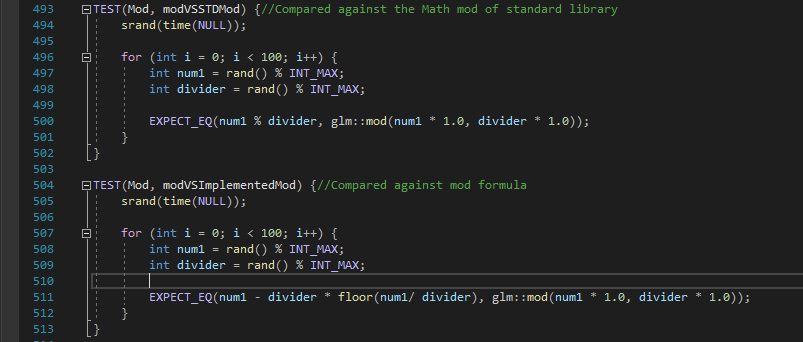
**Unit Testing:**

Within GLM, Core features, I focused on testing the **Integer** module and most of the **common functions** module. All tests are in <https://github.com/csun-comp587-s20/GLM/blob/roee/glm-test/glm-test/test.cpp>

Since this is a header only library, code changes and coverage testing were not accessible. Unit tests were written for every function tested within those modules and verified expected behavior matches the API spec. GLM is a math library, and therefore it is focused on one element which is crucial to its credibility – **Precision**.

* As suggested in my proposal, the **precision** of the results was tested, and compared to other known functions which implement similar algorithms. For example, for modulus, the GLM functionality was compared against both the C++ standard modulus, and the modulus formula:

x – y \* floor(x/y).

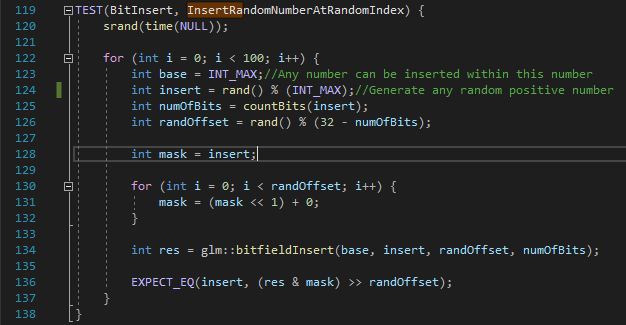


Additionally, edge cases were tested, like manipulations on 0,1, negative numbers, and streching the limit to all 32 bit integers. To do that, the tests used a random number between 0 and INT\_MAX.

* Following our discussion regarding my proposal, the speed of the executions is hard to test and compare against, and therefore was not focused on.
* To compensate for the inability to test for code coverage, I executed some of the functions for all positive integers (0 to INT\_MAX) to verify that those tests were indeed valid and covering all options. This process took hours and therefore the current code contains about 100 random numbers for those tests. This could easily be reverted back to traverse all positive ints. Additionally, some functions had parallel algorithms implemented in C++ standard library or known formulas that could be compared with. This served as “insurance” that GLM functionality was indeed precise.

**Automated Testing:**

As mentioned, GLM is used for all sorts of manipulations on mostly positive numbers. Testing all positive numbers take hours and was tried for some tests. However, to get a report in a reasonable amount of time, the automated test focused on a set of 100 random numbers between 0 and INT\_MAX (2,147,483,647). For some tests, which took several arguments, all arguments were randomized to ensure larger input set. For example, the function bitFieldInsert takes a base (number to which will be inserted), insert (the number to insert), offset (the offset in bits of where to start inserting the number), and bits(how many bits to insert). In this case, both the inserted number and the offset were randomized to test billions of different possible cases as shows below:



**Lessons Learned:**

* Testing a header only library with no access to the code fits black box testing well, but not necessarily allows unit testing with confirmed coverage.
* Working with numbers only, limits the input set and makes automated testing more efficient in covering test cases, though less interesting.
* Given the fact that GLM is a header library and code coverage is not possible, utilizing Dafny to compare GLM functionality to known formulas could have added another source of credibility.
* Testing the speed of processing is difficult. Given more knowledge about the system, I could have possibly integrated it with some sort of a video game and see how using GLM functionality compares to other libraries used by that game. However, the integration would have probably taken most of the effort, and add potential flaws that do not have to do with the library itself.

**Flaws Found:**

GLM is a mathematical library, and as such it is expected to support negative numbers or at least flag results involving negative numbers in case they are out of scope for a certain calculation. Many of the functions tested do not support negative numbers and return unexpected results when used. An example for which negative numbers returned such unexpected result and in fact did not follow the library spec is the function bitFieldExtract (This function’s purpose and details are discussed above, under the automated testing section). An issue arises when using bitFieldExtract, specifying negative offset or bits. In such case, the insert should result in undefined result (as shown in the screenshot below). However, a calculation has been made and returned a value.

