PrintFloat in D

from Bernhard Seckinger

PrintFloat is a replacement for the C function snprintf, which currently is used inside D's format command. This first version of PrintFloat only replaces the 'f' (and the 'F') qualifier for float and double.

About 6,000,000,000 tests have been run on floats and about 260,000,000 tests have been run on doubles. All tests compared the result of the new function with the old function. There were two differences found: a) A bug in the old function for large output and b) rounding differences which are intentional, because the new version uses a different tie for rounding mode 'roundToNearest'.

1,600,000,000 tests have been run (half float, half double) to compare the speed of both functions. While on first sight, it looks like the new algorithm outperforms the old one in most cases, having a closer look reveals that the new algorithm can be improved for large exponents and for very small exponents. The new algorithm can be much better optimized by the compilers though, which straightens the results.

All four rounding modes recommended by the IEEE754 standard are implemented. The mode 'roundToNearest' leaves two possibilities for handling ties, from which the more human like version is choosen. The other version is implemented too, but commented out.

Internationalization is not implemented as it has lots of negative effects and should therefore be done by a separate modul.

Ryu and other fast algorithms have been rejected, because they cannot be used for printf like functions due to a different design goal.

Finally, there is still a lot to do, starting with the implementation of the 'e' specifier and ending with adding support for reals and CTFE.

1 Correctness

From thoroughly going through the code I think it's correct, but I cannot prove that formally. Therefore I fall back to testing.

Unfortunately it's impossible to test all numbers with all variants of format strings and all rounding modes, because there are too many of them, approximately 5×10^{16} float tests, when width and precision are restricted to be at max 200. And about 10^{27} double tests, when width and precision are restricted to be at max 500. So I resorted to testing some subsets.

I checked three such subsets: The first being completely random, the second checking for all combinations of all flags, rounding modes, and a selection of widths, precisions and numbers and the third being a test of all float numbers for two format strings and of a fixed subset of all double numbers for those format strings.

1.1 Random Test

1,000,000,000 tests have been run on float numbers.

For every test a random number has been choosen uniformly, a random rounding mode has been set and a random format string has been created. That format string could include every of the five flags (dash, plus, space, zero and hash) with a 50% chance. The width and the precision were independently either empty (1% chance or taken uniformly from between 0 and 200). Finally one of the two qualifiers f and F was choosen randomingly.

For all tests, the result of the old function has been compared to the new function. No difference was found in the results.

The same test was applied with the rounding rule 'Round to nearest, ties away from zero'. Here 99.95% of all tests where correct. The reason is, that the old function does not support this rule.

The same tests where used for doubles, but this time only 100,000,000 test have been run. Again there was no difference found and with the alternative rounding rule 99.99% of all tests where correct.

1.2 All Flags Test

For this test all combinations of all flags, all rounding modes and both qualifiers (f and F) where used. Every one of these combinations was combined with all possibilities of a certain set of width, of precision and of a number.

This made up for a total of 1,244,160 checks for floats and 2,385,152 checks for doubles. Again the result of both function was compared and for floats no differences were found. For doubles only 1,934,464 tests where correct. The 450,688 different results appeared, because the old function is limited to 500 characters of output, while the new function can handle arbitrary lengths. In other words: This is due to a bug in the old function.

1.3 All Floats Test and Some Doubles Test

For the floats all possible bit patterns where scanned and the result of both functions compared for the two format strings '%f' and '%100.40f'. In both cases no differences where found.

For doubles it is not possible to scan all bit patterns, because there are too many of them. Therefore only a selection has been scanned, namely all bit patterns, where the first half is identical to the second half and where this half interpreted as a uint was dividable by 77, making up for 55,778,797 checks. Again all these numbers where tested for the two format strings '%f' and '%100.40f'. And again in both cases no differences where found.

1.4 Conclusion for correctness

Taking aside the rounding issue, which is addressed in the section about rounding, all tests confirm, that both functions are identical for most cases. The only exception is, when the output is longer than 500 characters. In that case the new function is correct, while the old function gives a truncated result.

2 Speed

I had no opportunity to check the speed on any other computer than mine, which has a 64bit intel compatible processor and debian as operating system.

Again it's not possible to compare all combinations of all format strings and all numbers for speed. Therefore I used a random sample to determine the speed.

For comparing the algorithms, the value of the exponent is crucial. Therefore, the diagrams in the rest of this section distinguish between the different exponents of the number.

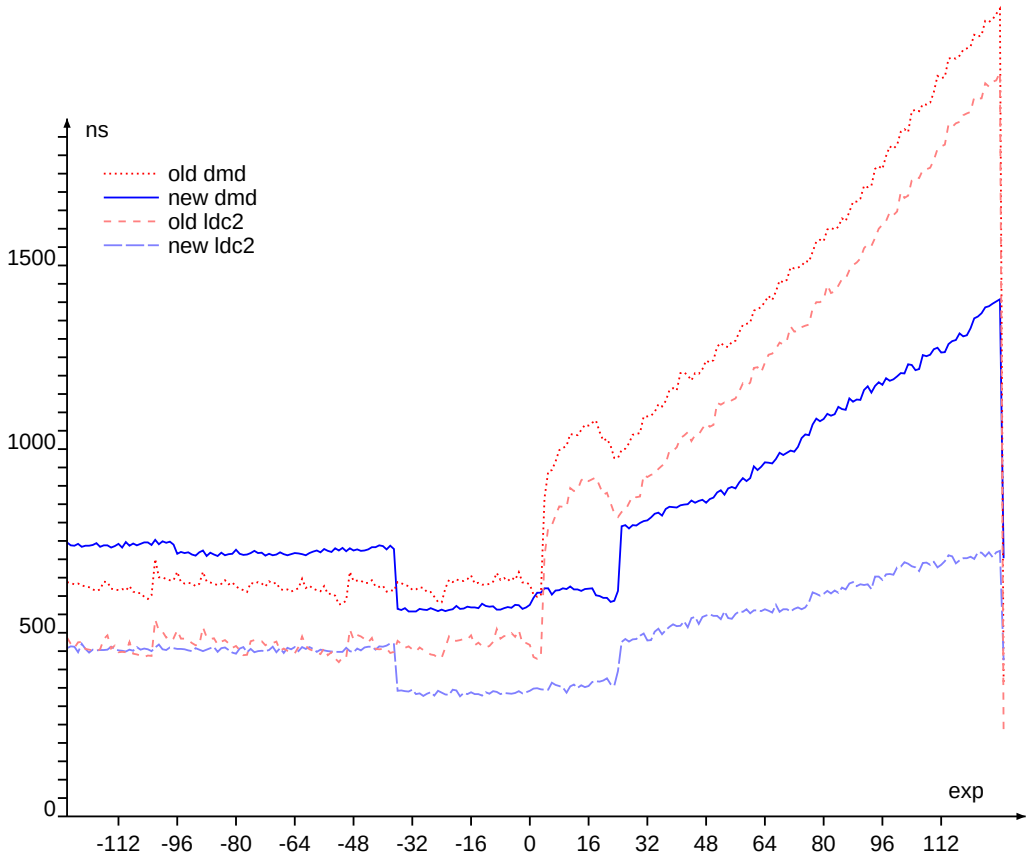
2.1 Overall test for float

The format string, the rounding mode and the number have been choosen randomingly, like stated in the Random-Test-section about correctness above.

100,000,000 tests have been run with `dmd -O -release` and with `ldc2 -O3 -release` and the execution time of both functions has been compared.

With dmd on average the old function took 1040ns while the new function took 827ns. In 59.1% of all tests, the new function was faster than the old function.

With ldc2 on average the old function took 876ns while the new function took 488ns. In 75.9% of all tests, the new function was faster than the old function.



As can be clearly seen, the new algorithm is superior for numbers with $exponent \geq -39$. The remaining numbers, which are numbers close to zero, are slower when dmd is used and about the same, when ldc2 is used.

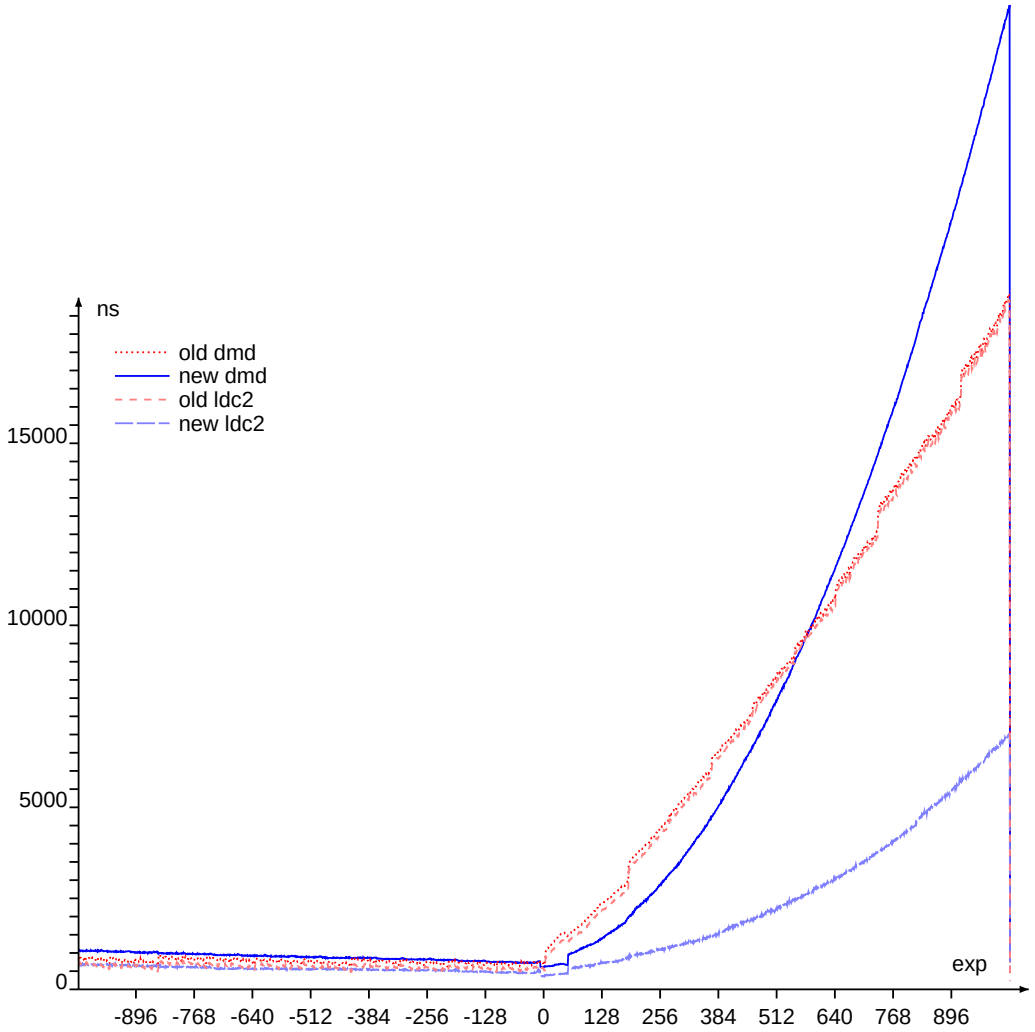
I see the possibility to improve the speed for numbers close to zero by estimating the number of zeros directly after the dot, but the implementation of that is tricky and needs to be done with great care to avoid a speedlos at other places, and therefore is left for later.

2.2 Overall test for double

The setting for the double tests is the same as for the float test.

With dmd on average the old function took 4908ns while the new function took 5402ns. In 31.1% of all tests, the new function was faster than the old function.

With ldc2 on average the old function took 4748ns while the new function took 1640ns. In 86.0% of all tests, the new function was faster than the old function.



Here, clearly the numbers with large exponent dominate. It's obvious, that the old algorithm is almost linearly growing, while the new one is quadratic. For doubles, this is still ok, because ldc2 manages to optimize enough. But the new algorithm cannot be used for reals anymore.

Again I see the possibility to improve this algorithm. This time it should take advantage of knowing, that the binary representation of the large number has a lot of zeros at the end. Again the implementation is tricky and needs to be done with great care and therefore is left for later.

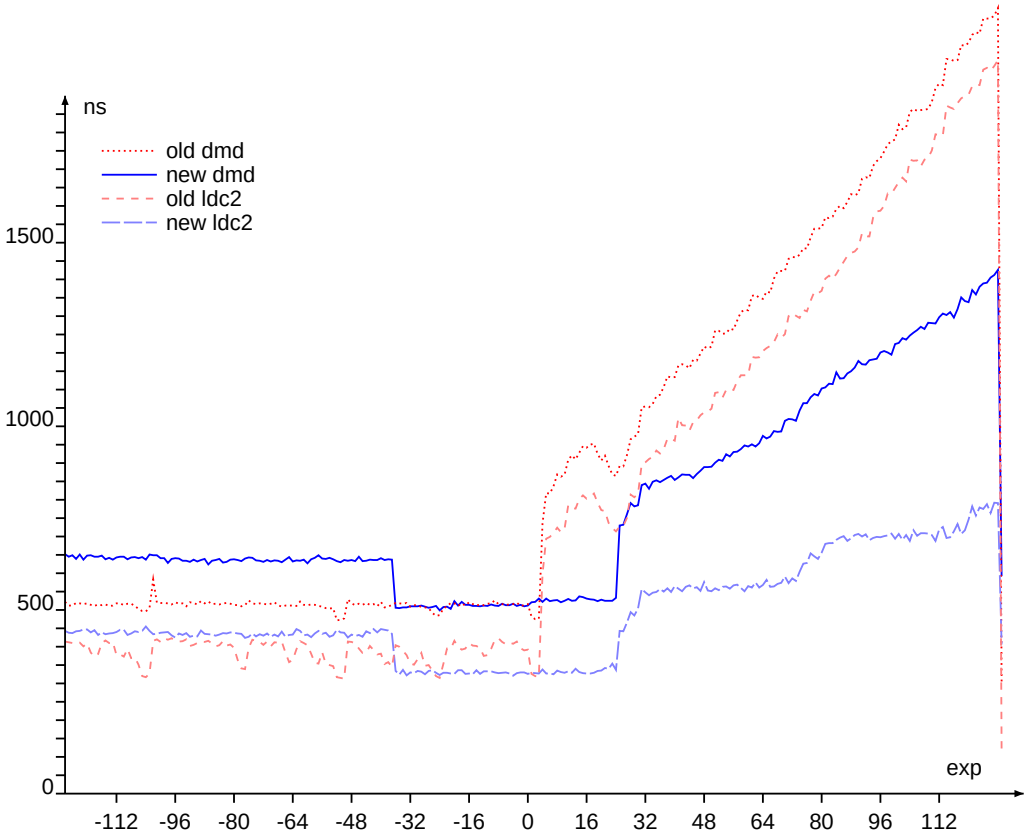
2.3 Special tests for float

Three special tests have been used with fixed format specifier and only default rounding mode. The used format specifiers where `%f`, `%.100f` and `%50.25f`. Again 100,000,000 tests have been run on each variation.

2.3.1 %f

With dmd on average the old function took 953ns while the new function took 786ns. In 50.9% of all tests, the new function was faster than the old function.

With ldc2 on average the old function took 814ns while the new function took 491ns. In 60.0% of all tests, the new function was faster than the old function.

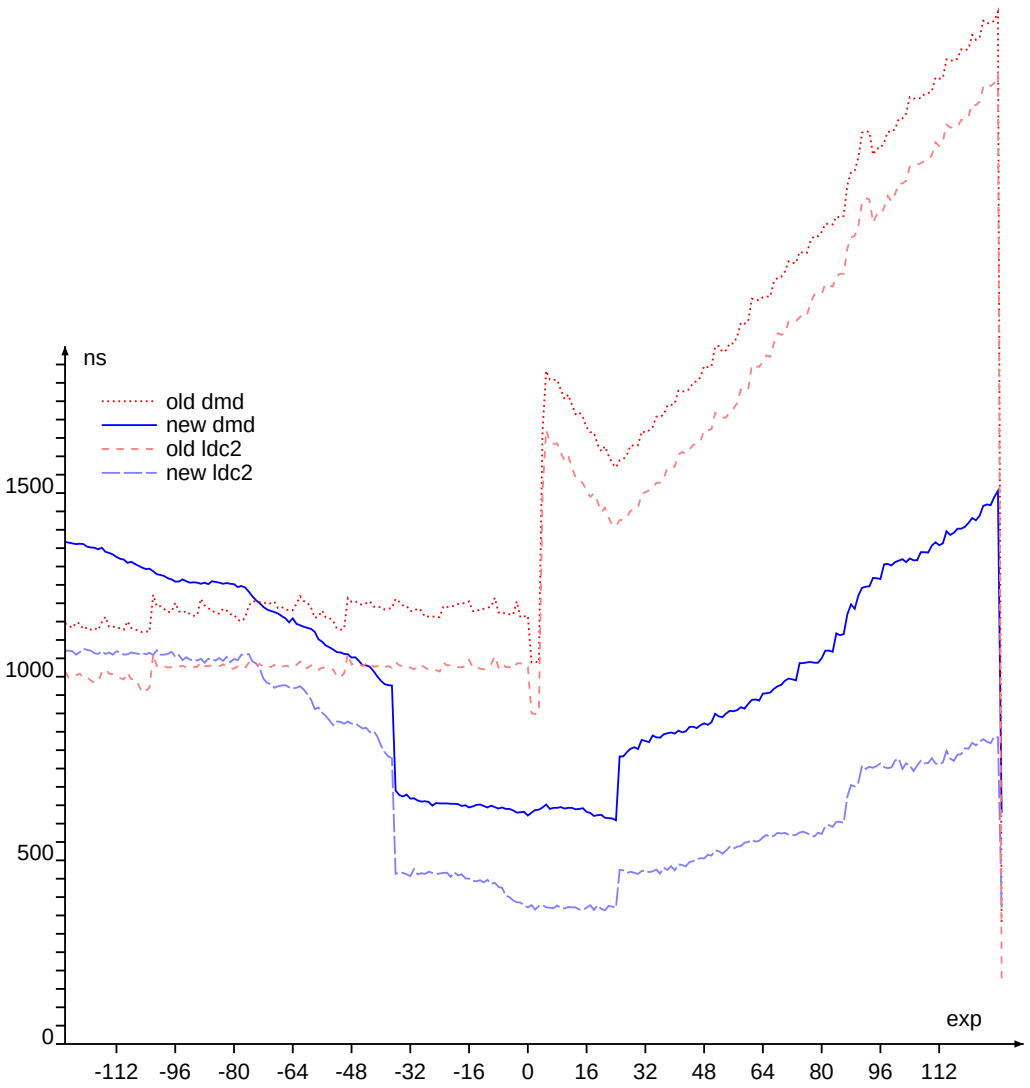


The result looks similar to the general case.

2.3.2 %.100f

With dmd on average the old function took 1628ns while the new function took 1025ns. In 76.2% of all tests, the new function was faster than the old function.

With ldc2 on average the old function took 1465ns while the new function took 705ns. In 80.9% of all tests, the new function was faster than the old function.

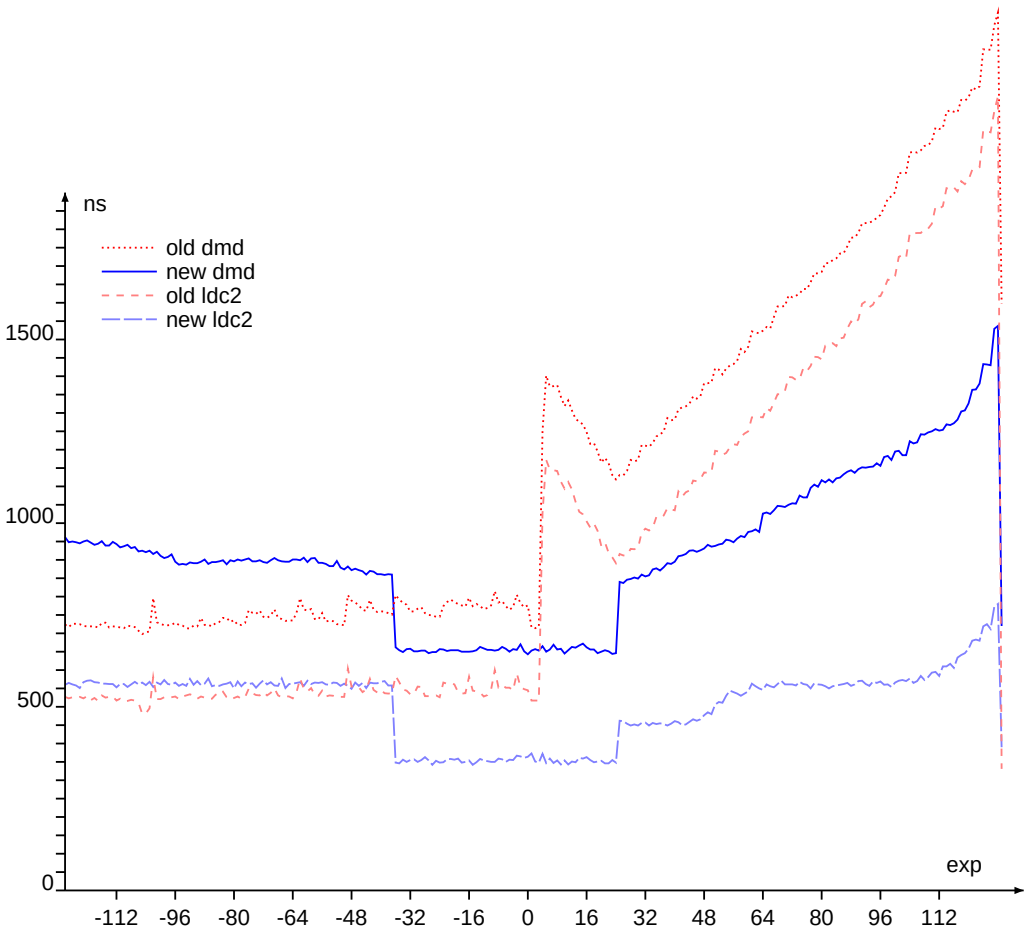


The new function here is clearly superior. Only for numbers with very small exponent, the duration seems to increase somewhat linearily while the old function manages to stay constant. This will probably be overcome by the improvements mentioned above.

2.3.3 %25.50f

With dmd on average the old function took 1171ns while the new function took 918ns. In 63.5% of all tests, the new function was faster than the old function.

With ldc2 on average the old function took 947ns while the new function took 508ns. In 67.5% of all tests, the new function was faster than the old function.



Again, this looks similar to the general case. The old function has a strange peak for numbers width exponent little above zero. This peak appeared in former diagramms too, but in a less extreme way.

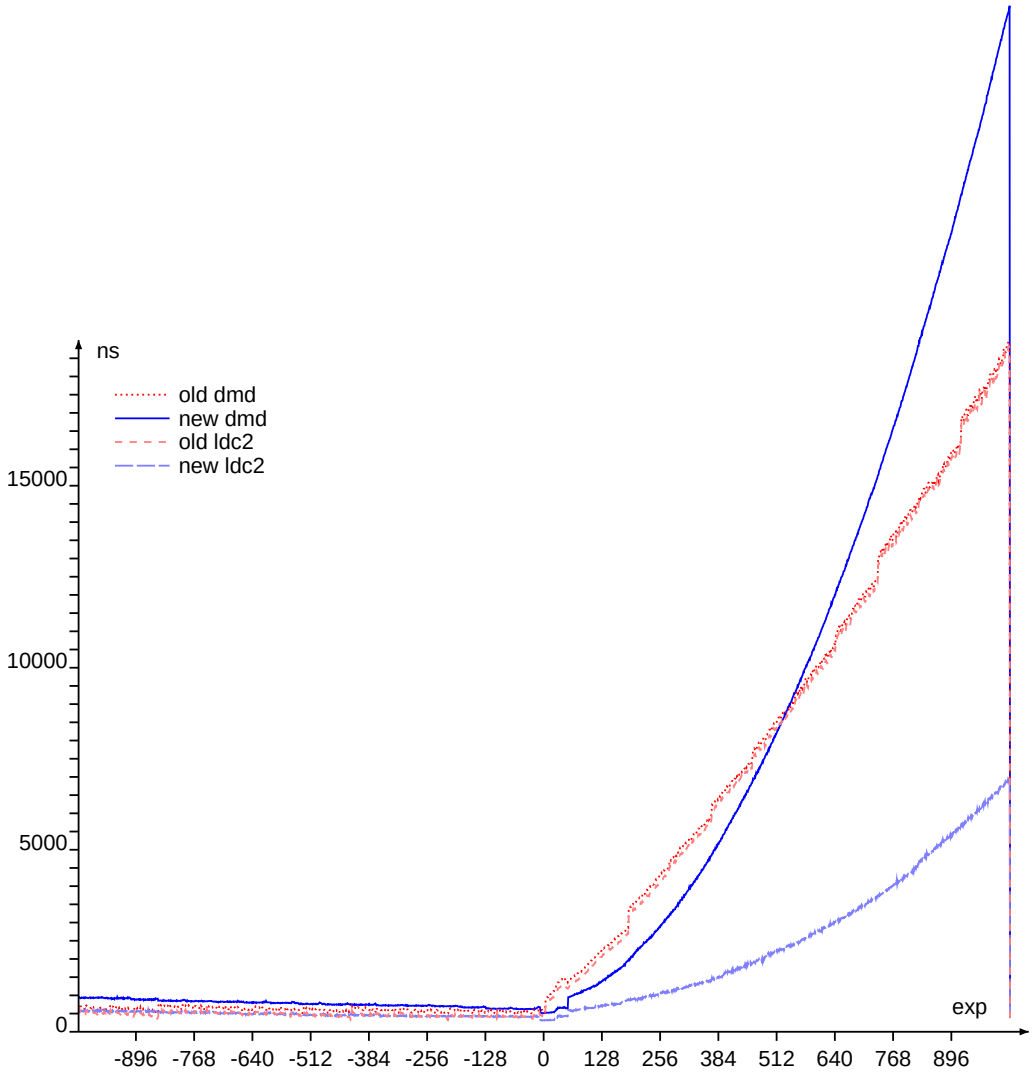
2.4 Special tests for double

The same three tests have been applied for doubles.

2.4.1 %f

With dmd on average the old function took 4770ns while the new function took 5524ns. In 26.8% of all tests, the new function was faster than the old function.

With ldc2 on average the old function took 4622ns while the new function took 1582ns. In 64.6% of all tests, the new function was faster than the old function.

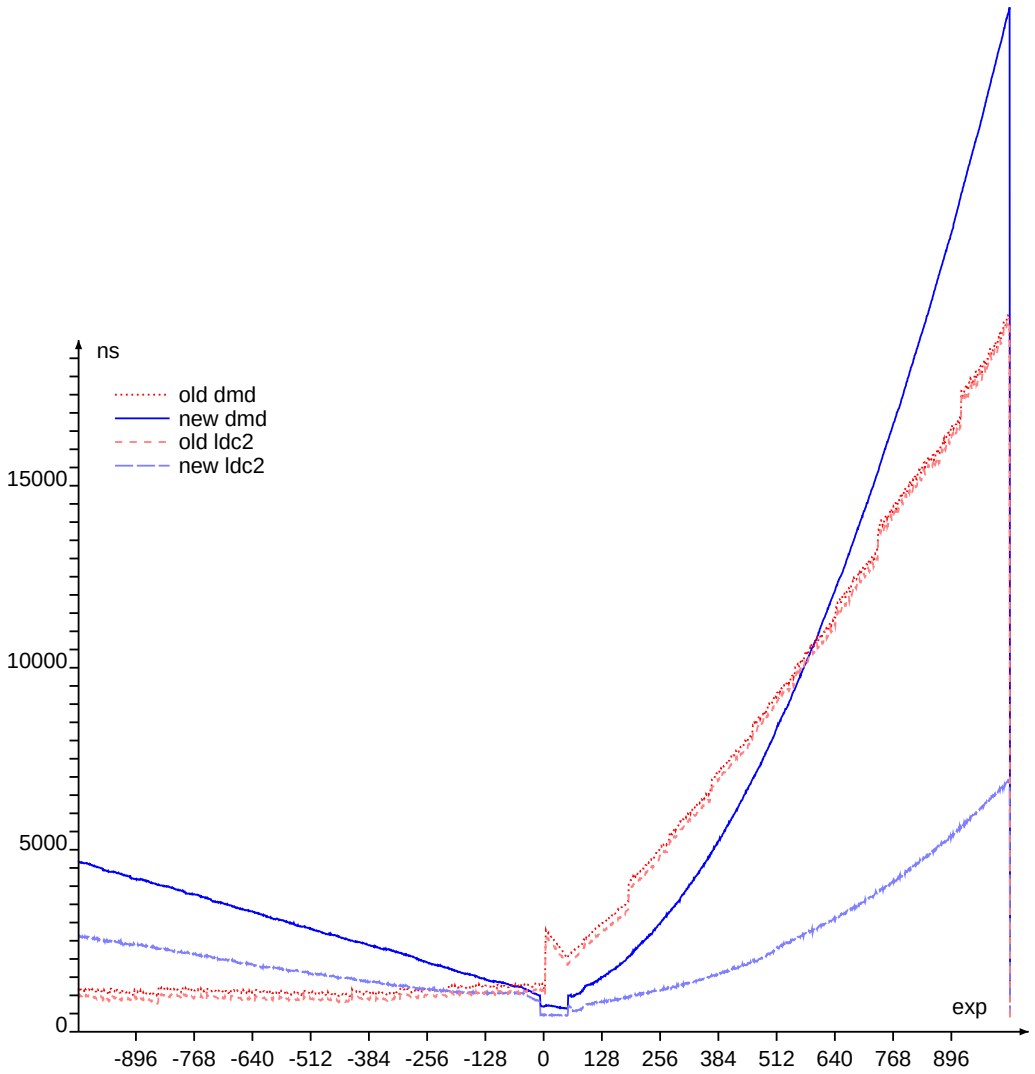


The result looks similar to the general case.

2.4.2 %.100f

With dmd on average the old function took 5402ns while the new function took 6584ns. In 31.9% of all tests, the new function was faster than the old function.

With ldc2 on average the old function took 5213ns while the new function took 2207ns. In 55.9% of all tests, the new function was faster than the old function.

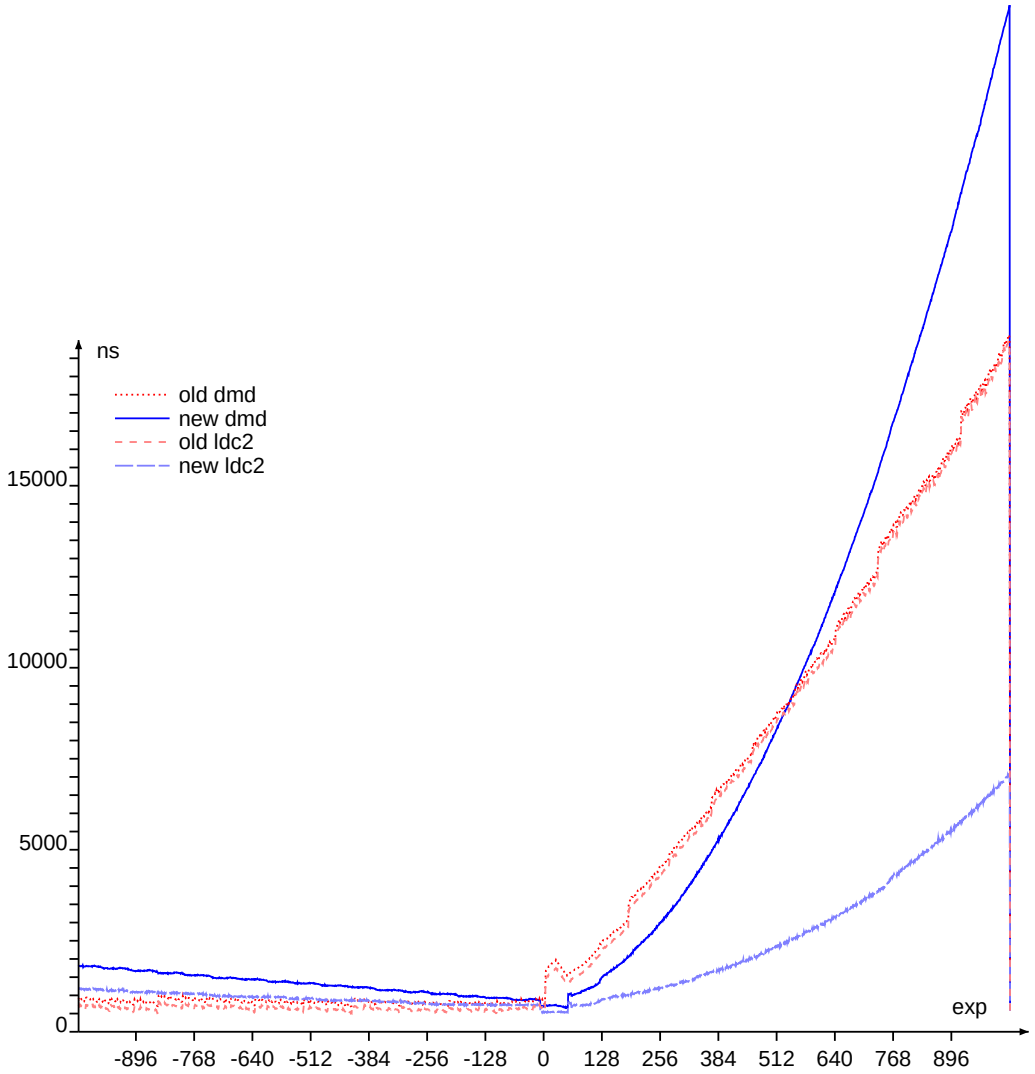


This again looks similar to the general case. The only clearly visible difference is the linear growth for small exponents, that allready was found for floats in that case.

2.4.3 %25.50f

With dmd on average the old function took 4985ns while the new function took 5836ns. In 27.5% of all tests, the new function was faster than the old function.

With ldc2 on average the old function took 4795ns while the new function took 1873ns. In 50.5% of all tests, the new function was faster than the old function.



Again, this looks similar to the general case.

2.5 Conclusion for speed

Both algorithms are in the same order of magnitude. The new one is clearly superior, when compiled with ldc2 and optimization turned on. In general, the new algorithm is better for numbers with exponent close to zero and tends to get worse, the farther away from zero the exponent is. While this is also true for the old algorithm for large exponents, for small exponents it isn't.

3 Rounding

According to IEEE754, decimal representations of a floating point number should obey the current rounding mode. There are four rounding modes available: RoundToZero, RoundUp, RoundDown and RoundToNearest. While the first three modes are unproblematic (both functions handle all of them correctly), IEEE754 allows for the last one two versions, called 'Round to nearest, ties away from zero' and 'Round to nearest, ties to even'. The old function uses the second version, while the new function contains both versions.

My personal oppinion is, that 'ties away from zero' is more natural for human readable output, than 'ties to even' and I therefore suggest to change this.

It has been argued, that 'ties to even' is more suitable, if the results will be read in again, used in some calculation and printed out again, and this repeated times. This is true, but in my oppinion it is a misuse of the format function. Such algorithms should use a binary format or use the 'a' qualifier instead of the 'f' qualifier. The 'a' qualifier produces a 100% rounding free result.

4 Internationalization

It has been suggested, that the functions should obay the locales. For me it's unclear if the old function really does so. From looking into the source code it seems to me, that if it does, it only consideres to replace the dot by some other symbol. All other ways of localisations are ignored.

The new function ignores internationalization alltogether. This is intentional. In my oppinion internationalization is a complicated thing and should be left to a specialized module, that can take the results of format and make the necessary corrections. All else will have negative effects on the printFloat, like blowing up the code, making it impure and probably even unsafe, slower and the risk of bugs rises significantly. To avoid these problems, I left out any support for internationalization.

5 Why not Ryu or Grisu?

Several people pointed out, that there are very fast algorithms arround for conversion from binary floating point numbers to decimal representation. Why do I not use these?

Well, mainly, because they don't work here. Those algorithms are designed with some other design goal, namely to produce an as short as possible representation of a number, that, when read in again, produces the same binary bit pattern again.

Contrary to that, printFloat get's a user defined value (precision) that exactly tells how many digits should be printed. For example the number 0.123456789f is internally represented by the bit pattern 00111101111111001101011011101010. Which is $2^{-3} \times 1.11111001101011011101010_b = 0.12345679104328155517578125$.

All numbers between 0.123456787317991256713867 5 and 0.12345679476857185363769 53125 are represented by that bit pattern. From all those numbers ryo chooses the shortest, here 0.12345679.

But `format!"%.15f"(0.123456789f)` should produce 0.12345679104328. With ryo it's not possible to add the remaining 6 digits[1] and figuring them out would be much more expensive than just not using ryo from the very beginning. The same holds for the grisu variations.

6 Roadmap

There are several ways to continue from here:

- Add 'e', 'g' and 'a' specifier.
- Add optimization for small exponents.
- Add optimization for large exponents.
- Make the function(s) CTFEable.
- Add support for reals.

[1] In some cases, ryo might even round the number up, which means, that some digits at the end of the number may come out wrong and need to be corrected too.

Adding the ‘e’ and ‘g’ specifier should be rather straight forward. Adding the ‘a’ specifier needs a new function, but this will be much easier to implement, because binary and hex representation are much closer to each other, than binary and decimal.

It’s unclear how difficult it will be to add the optimizations and if they really yield some benefit. But this should be done, before reals are implemented, because they will mostly benefit from these optimizations.

For me, it’s also unclear, how much effort needs to be taken, to make the function CTFEable. But as this tends to make the function more ugly (in my oppinion), I tend to postpone this till the end.

Support for reals would be easy, if the datatype ucent would be available. With this type, just ulong would need to be replaced by ucent and the extraction of sign, exponent and mantissa needs to be added. Without, a workaround needs to be implemented which might be much work.