**Precision of Floating Point of IEEE 754 Compliance  
for upper-level implementations**

|  |  |
| --- | --- |
| **Katarzyna Niemyjska 242560** | **Martin Machnikowski 188138** |
| **Urszula Kulon 236343** | **Mieszko Wawrzyniak 242563** |
|  | **Patryk Szwed 242363** |

Who are these who float along like a cloud, who fly like doves to their shelters?

Isaiah 60:8

**ABSTRACT**

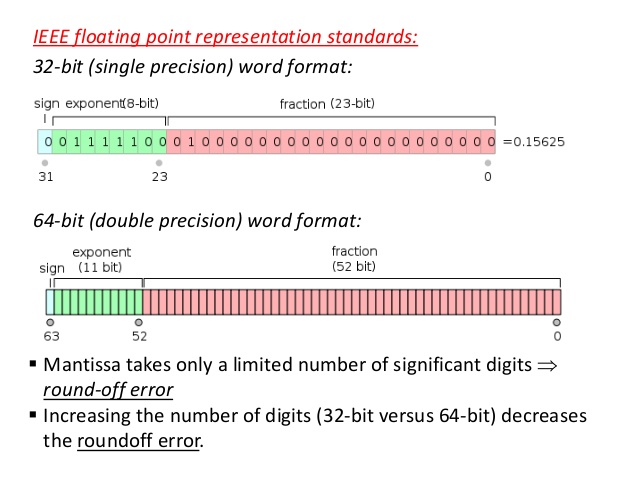
**Investigations on accuracy of floating point formats supported by today’s computer architectures and their upper-level implementations.**

# ****Introduction****

The IEEE 754 Standard for Binary Floating-Point Arithmetic (IEEE 754-1985), which was widely adopted – it’s supported by most today’s processors and is implemented in most of programming languages, introduced a standardization how arithmetic results should be approximated in floating point. In order to use computers in any scientific application, we must consider many aspects of floating point behavior, since we are working with inexact results which can affect accuracy. Floating point encodings and functionality are defined in the IEEE 754 Standard last revised in 2008.

The computer processor has a built-in support for integers (int) and real (float) numbers. Real ones can be written as a decimal fraction so as two integers. Real numbers (float) are stored as three numbers:

***float = mantissa \* base exponent***



However, not all numbers can be written in that way. The range of floating point numbers, as defined in the IEE 754, is from *1.17549 \* 10-38*to *3.40282 \* 1038*. A consequence is that, in general, the decimal floating-point numbers that are entered are only approximated by the binary floating-point numbers actually stored in the machine. Considering the fraction 1/3, it can be approximated as a base 10 fraction:

0.3  
or,

0.33  
or,

0.333

and so on. Regardless of how many digits are to be written down, the result will never be exactly 1/3, but it will be an increasingly better approximation of 1/3. In the same way, no matter how many base 2 digits are to be used, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction

0.0001100110011001100110011001100110011001100110011...

Stopping at any finite number of bits, and gives an approximation. For most calculations such inaccuracy is irrelevant. For instance, when the area of a circle is calculated, in most cases, it’s enough to approximate that π is 3.14. However, it’s important to remember that in further places there may be digits different from zero, which are involved in all mathematical operations, and multiplied can give unexpected outcomes. The more mathematical activities are performed, the greater the divergence from the real result might be. Adding big numbers to smaller ones should be avoided, since the result will be much more inaccurate than in the case of multiplying them or adding close ones.

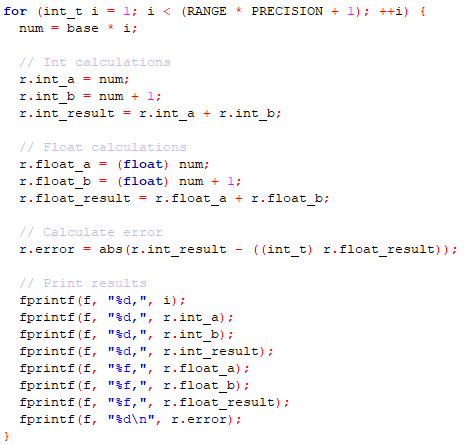
# Plan of the experiment

Hypothesis: The approximation error of real number addition differs depending on the magnitude of the number given, and depends on which of the 32 and 64 bit basic binary floating point format is used.

To test this statement, we generate two arrays, consisting of 1000 integer and real numbers of the same base each, one for float addition and a second one for double-precision addition. Then we use an upper-level implementation of addition to generate a sum of each of before mentioned objects, and we compare the results between integer and float (or integer and double respectively) to check, whether our hypothesis holds. We suspect, that the closer we are to the base number, the lower the discrepancy between results will be.

# The Experiment and results

Upper-level implementation of floating-point addition is written in C using GNU GCC Compiler in the Code::Blocks 16.01 (build wx2.8.12) IDE.



Example of a code used to generate test samples

All code files and generated workbooks are appended.

All charts, and the linear trend line were made using Microsoft Excel 2016.

# Discussion and conclusion

As the hypothesis states, the bigger the numbers which are added, the bigger is the discrepancy between integer and float/double addition. The closer our added number is to a multiplication of given base, the lower the error is (dips on the charts). We can also see, that using 64-bit approximation until the number 256 the inaccuracy is nonexistent, and subsequently steadily steps up with every power of 2 bigger than 256. In comparison, in case of the 32-bit approximation, the error shows starting with the 8 number, and likewise escalates with every next power of 2.

Considering the angle of calculated trend lines, we can conclude that using double (64-bit) word format helps remarkably with minimizing the addition error in comparison to the float (32-bit) word format. For instance, adding numbers of comparable magnitude, in float format gives us a 65 divergence, while using double format gives us only a 5 point difference from integer addition.

In conclusion, as the hypothesis states, the approximation error of real number addition differs depending on the magnitude of the number given, and depends on which of the 32 and 64-bit basic binary floating point format is used. It is advised to carefully consider, whether to use which encoding depending on the task given, since using 64-bit representation uses more memory, and since not in every application such precision is critical thus using it might not be optimal.

# Bibliography

1. ANSI/IEEE 754-1985. *American National Standard — IEEE Standard for Binary Floating-Point Arithmetic.* American National Standards Institute, Inc., New York, 1985
2. IEEE 754-2008. IEEE 754–2008 Standard for Floating-Point Arithmetic. August 2008.
3. <http://www.ristel.pl/~mielecki/introduction-to-computer-systems.html>
4. William Stallings, Computer Organization and Architecture. Designing for Performance.
5. [*Knuth, Donald Ervin*](https://en.wikipedia.org/wiki/Donald_Ervin_Knuth) (1997). "Section 4.2: Floating-Point Arithmetic". [*The Art of Computer Programming*](https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming). 2: Seminumerical Algorithms (3 ed.). [*Addison-Wesley*](https://en.wikipedia.org/wiki/Addison-Wesley). pp. 214–264. [*ISBN*](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [*0-201-89684-2*](https://en.wikipedia.org/wiki/Special:BookSources/0-201-89684-2).
6. Kernighan B. W., Ritchie D. M.: C Programming Language (2nd Edition), Prentice Hall Software Series