Methodological Aspects of Studying the Accuracy of Computer Calculations in Applied Problems

Oleh Vietrov[1], Olha Trofymenko[2], Vira Trofymenko[1] and Maxim Hryhorov[1]

[1] *Vasyl' Stus Donetsk National University, 21, 600-richchia str., Vinnytsia, 21021, Ukraine*
[2] *University of Connecticut, 341 Mansfield Road, Storrs, CT 06269, USA*

Abstract

The study is dedicated to considering some examples of computer calculations that are completely correct from the point of view of the standards of computer work with numbers and arithmetic operations, the result is incorrect to a human performer. It's demonstrated that mathematically correct calculations are often impossible to reproduce correctly within the IEEE-754 standard due to the fundamental limitations of the memory allocated to represent a specific number. The authors focused on the study of the specified problem using the example of the numpy module, as one of the main software tools for modern scientific calculations. The work has methodological value in the professional training of specialists in Computer Science.

Keywords

Computer calculations, IEEE-754 standard, numpy

1. Introduction

The problem of reliability of computer calculations is one of the fundamental problems of Computer Science [1], as it lies at the intersection of applied mathematics on the one hand, and physical and technical limitations of computer technology on the other. The avoiding the fundamentals of binary arithmetic and the principles of the standard for representing numbers in computer memory can significantly affect the correctness of both theoretical research (if computer technology was used for modeling) and simply the quality of the product when developing application programs.

The problems that can arise due to incorrect representation in computer memory are well known. Sometimes they are subtle problems that do not have a significant impact in a qualitative sense on a particular result. Therefore, in application activities, floating-point computing problems and features of IEEE-754 standard are often overlooked by software developers as well as academic researchers. However, there are notorious cases when careless handling of computer representation of numbers led to serious tragedies. A well-known example of the tragic consequences of such problems is "The Patriot Missile Failure" [2-3]. The main problem was that the time value in tenths of a second was multiplied by a factor of 0.1 to get a representation of time in seconds. This calculation was done using a 24-bit fixed-point register. The Patriot battery lasted about 100 hours and the accumulated error was about 0.34 seconds. One can also recall the precedent of "The Explosion of the Ariane 5" [4-5].As a result of computer calculations, a 64-bit floating-point number was converted to a 16-bit signed integer. The number was greater than 32,768, and so the conversion failed. An error in the system for calculating the horizontal velocity of the rocket relative to the platform resulted in damage estimated at 500 million dollars. Some examples of real losses caused by rounding errors or problems in representing numbers in computer memory are given in [6].

CMIS-2024: Seventh International Workshop on Computer Modeling and Intelligent Systems, May 3, 2024, Zaporizhzhia, Ukraine

o.vietrov@donnu.edu.ua (O. Vietrov); olha.trofymenko@uconn.edu (O. Trofymenko); v.trofimenko@donnu.edu.ua (V. Trofymenko); hryhorov.m@donnu.edu.ua (M. Hryhorov)
0000-0002-5125-9632 (O. Vietrov); 0000-0002-3925-5815 (O. Trofymenko); 0009-0005-8094-227X (V. Trofymenko); 0000-0001-6817-2624 (M. Hryhorov)

The limitation on the amount of memory allocated by computer technology for number processing imposes fundamental limitations on the possibilities and logic of the organization of computer calculations. It's also requires the involvement of specific mathematical research methods.

The standard and most popular tool for computer calculations, especially in machine learning tasks, is the Python programming language, in particular with its powerful extension NumPy module. It allows you to efficiently work with large multidimensional arrays and matrices, along with a large library of mathematical functions. The NumPy module provides much wider possibilities for working with mathematical functions and objects compared to the math module. The NumPy is an effective tool for providing complex mathematical calculations not only in the tasks of machine learning and data analysis, but is also an indispensable component in the implementation of a large volume of computer calculations in a wide class of modern engineering and scientific tasks [7-10].

The investigation considers the main aspects related to the problems of computer calculations within the IEEE-754 [11] standard, the main technical standard format for representing floating-point numbers (floating-point numbers) (discussing possible alternatives to the IEEE-754 standard, such as Unum [12] and Posit [13-14] go beyond the scope of the presented work).

The history of the development of computing and the basics of the theory of computing with floating-point numbers are described in [15-16]. Important related issues of floating-point computing are presented in [17-18]. Recently, the research in the direction of computer-assisted proof for various classes of mathematical problems [19] is more developed. In this work, the authors touch on possible incorrect cases of the possible use of proof calculations in case of failure to take into account certain features of the representation of numbers in the computer memory.

The presented work is a continuation of the authors' research [20-21]. Previously, the authors investigated rather complex cases of violation of the correctness of computer calculations, the example of Ramp [22] and the sequence of Müller [15]. The works of [20-21] investigated the mathematical aspects of computer calculations, in particular the convergence and stability of computer calculations. The authors focused more on the methodological principles of teaching the topic of accuracy of calculations for the Computer Science specialty students in the presented work. Based on the teaching experience, the authors believe that a detailed study of the problem of the incorrectness of computer calculations on trivial examples (rather than specially designed ones) will allow students to take a more serious and attentive approach to the problems of computer calculations.

2. Methods

Let's consider the numeric data types of the NumPy library, one of the most popular Python libraries for scientific computing. Figure 1 shows the hierarchical diagram of the numpy.generic data class, the base class for numpy scalar types. We not consider components of numpy.generic like nympy.bool_, nympy.object_, nympy.datetime64 or nympy.flexible.

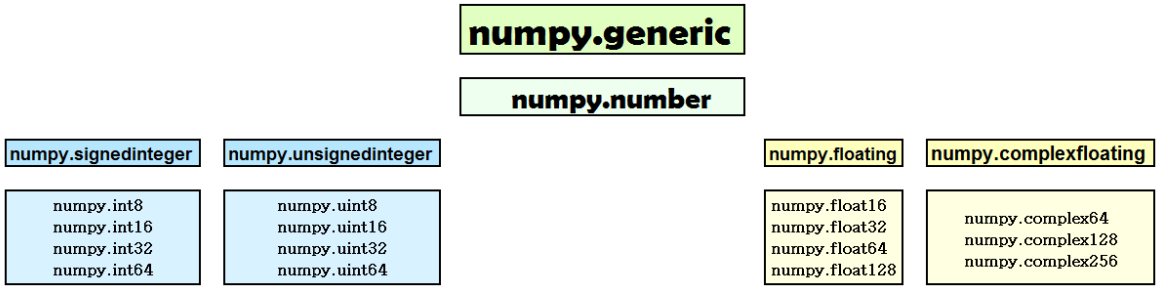

Figure 1: Numpy number types

Floating point numbers according to the IEEE 754 standard are presented in the form

$$number = (-1)^s \times 1.M \times 2^E$$

where S is number sign (signum), M is mantissa, E is number order (exponent). Figure 2 shows the distribution of bits of a single-precision number in the binary32 format (the most significant bit is the sign, the order of the number is preserved using 8 bits, 23 bits are allocated to the mantissa – a total of 32 bits).

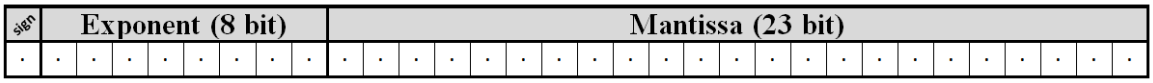

Figure 2: Single precision number (IEEE-754)

The value ranges of the types of floating point numbers are presented in Table 1.

Table 1
Floating-point types

Numpy type	Alias	Precision floating-point number type
half	float16	sign bit, 5 bits exponent, 10 bits mantissa
single	float32	sign bit, 8 bits exponent, 23 bits mantissa
double	float64	sign bit, 11 bits exponent, 52 bits mantissa
longdouble	float128	sign bit, 15 bits exponent, 64 bits mantissa

The mantissa is normalized (the integer part is equal to one), as we can see from the formal representation of the number n. To calculate the order, a constant shift exponent is used, which for single-precision numbers is 127 (or 01111111 in binary code).

Let's consider for the clarity, what will happen to the representation of the rational number $1/3$ according to the described principles of IEEE 754. Mathematically, in the decimal and binary number systems, we have

$$1/3 = 0.(3)_{10} = 0.(01)_2.$$

For a single-precision number, we can write

$$1/3 = (-1)^0 \cdot 1.01010101010101010101011 \cdot 2^{-10},$$

i.e S = 0 (number is positive), M = 01010101010101010101011 (since the first discarded bit was a unit, according to the rules of the standard, we add 1 to the mantissa), E = -10 (the order is -2 in the decimal number system).

We have float32-approximation to the $1/3$ in the mathematical form

$$(0.01010101010101010101011)_2 = 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 1 \cdot 2^{-5} \ldots =$$
$$= \sum_{k=1}^{12} \frac{1}{2^{2k}} + \frac{1}{2^{25}} = (0.33333334326744079589843 75)_{10}$$

The obtained approximate value of the number is greater than the real value $1/3$. Everything is still correct in the mathematical sense

$$0.(01)_2 = \sum_{k=1}^{\infty} \frac{1}{2^{2k}} = \frac{1/4}{1 - 1/4} = 1/3$$

Let's consider an important concept of the theory of computer calculations – machine epsilon [23]. This is numerical value below which it is impossible to specify the relative accuracy for any algorithm that produces real numbers.

A close concept, but not identical, is the concept of machine zero. That is a numerical value with such a negative order that the computer perceives it as zero.

In fact, to get the machine epsilon value in Python you can use the function numpy.finfo() for specific floating point number formats. But it seems appropriate to give a software implementation (Listing #1) of calculating the values of machine epsilon and machine zero, in order to understand the essence of these concepts in a constructive way.

Listing #1

```
1       from numpy import *
2       def machineEpsilon(float_format = float):
```

```
3             iteration_count_eps = 1
4             mach_eps = float_format(1)
5             while   float_format(1) + float_format(mach_eps)\
6                   /float_format(2) != float_format(1):
7               iteration_count_eps += 1
8               mach_eps /= float_format(2)
9             return mach_eps, iteration_count_eps
10
11          def machineZero(float_format = float):
12            iteration_count_zero = 0
13            mach_zero = float_format(1)
14            while   float_format(mach_zero)\
15                  /float_format(2) != float_format(0):
16              iteration_count_zero += 1
17              mach_zero /= float_format(2)
18            return mach_zero, iteration_count_zero
```

Result:

```
Machine Epsilon
<class 'numpy.float16'>: (0.000977, 11)
<class 'numpy.float32'>: (1.1920929e-07, 24)
<class 'numpy.float64'>: (2.220446049250313e-16, 53)
<class 'numpy.longdouble'>: (1.084202172485504434e-19, 64))

Machine Zero
<class 'numpy.float16'>: (6e-08, 24)
<class 'numpy.float32'>: (1e-45, 149)
<class 'numpy.float64'>: (5e-324, 1074)
<class 'numpy.longdouble'>: (4e-4951, 16445)
```

We see that a specific machine epsilon, like a machine zero, is not fundamental. The order of given numbers is important. The value of the iteration_count_eps variable is the negative power order $\mathbf{2^{-iteration\_count\_eps}}$. As we can see, for the machine epsilon for the corresponding type of floating point numbers, the iteration_count value is one more than the number of bits allocated for the mantissa value for the corresponding type. This makes sense because it implies the precision limits for the corresponding numeric type.

As for the results for machine zero, the given values are the smallest numbers below which the computer already perceives as zero.

That is, in fact, the machine zero is a number $2^{-(\text{iteration\_count\_zero}+1)}$. It is obvious that it is possible to obtain the order of the machine zero without computer calculations using the formula

$$\text{mantissa} + 2^{\text{exponent}-1},$$

where the exponent and the mantissa determine the number of bits for the corresponding data type.

3. Results and Discussion

3.1. Restrictions on calculations in integers

Let's consider one interesting example of working with integers when overflowing a bit grid.

This problem itself is well known, but when training specialists, it is important to demonstrate that the results obtained are incorrect to a human programmer, but are nevertheless absolutely correct from the point of view of executing the program within the framework of existing standards. A simple example is the calculation of the value of the factorial of some natural number n. With implementing directly in Python (the factorial function from the math module), there are no problems with accuracy. Python implements the BigInt [24]

mechanism by default, that is, the value of an integer is not limited by the number of bits and can expand to the limit of available memory. At the same time, using the capabilities of the numpy module allows you to explore in more detail the nuances of factorial calculation for different types of integers.

Table 2 lists [25] some of the signed and unsigned integer types available in numpy.

The second column in Table 1 is the alias of the corresponding data types on the Linux x86_64 platform.

Table 3 shows the results of calculating the factorials of the number n in Python depending on the types of integers. The correct result is obtained using Python's standard int() (the BigInt concept mentioned above).

As we can see from Table 3, when calculating the factorial using the int32 type, starting with n = 13, the calculation result is incorrect. Using the int32 and int64 types, starting with a count of 21!, Python throws an OverFlowError exception.

Note that the numpy.intc and numpy.int_ types are analogues of the int and long long types in C, while the generation of the OverFlowError exception is a feature of the numpy module itself, while in C we would get the corresponding values

int factorial(21) = – 1195114496,
long long (21) = – 4249290049419214848.

We obtained a paradoxical result, that the product of natural numbers is equal to a negative number.

In integers in the int and long long formats, everything is calculated correctly.

Table 2

Integer types (Signed integer types)

Numpy type	Alias	Range of Values	
byte	int8	$-2^7 \dots 2^7-1$	-128 .. 127
short	int16	$-2^{15} \dots 2^{15}-1$	-32768 .. 32767
intc	int32	$-2^{31} \dots 2^{31}-1$	-2147483648 .. 2147483647
int_	int64	$-2^{63} \dots 2^{63}-1$	-9223372036854775808 .. 9223372036854775807

Integer types (Unsigned integer types

ubyte	uint8	$0 \dots 2^8-1$	0 .. 255
ushort	uint16	$0 \dots 2^{16}-1$	0 .. 65535
uintc	uint32	$0 \dots 2^{32}-1$	0 .. 4294967295
uint	uint64	$0 \dots 2^{64}-1$	0 .. 18446744073709551615

The real value is 21! in decimal and binary number systems is

$21! = (51090942171709440000)_{10}$,

$21! = (1011000101000001110111110100110110101110001100010000000000000000000)_2$

Table 3

Factorial calculation depending on integer type

n	Correct value n!	int32(n!)	int64(n!)
1	1	1	1
2	2	2	2
3	6	6	6
4	24	24	24
5	120	120	120
6	720	720	720
7	5040	5040	5040
8	40320	40320	40320

9	362880	362880	362880
10	3628800	3628800	3628800
11	39916800	39916800	39916800
12	479001600	479001600	479001600
13	6227020800	1932053504	6227020800
14	87178291200	1278945280	87178291200
15	1307674368000	2004310016	1307674368000
16	20922789888000	2004189184	20922789888000
17	355687428096000	-288522240	355687428096000
18	6402373705728000	-898433024	6402373705728000
19	121645100408832000	109641728	121645100408832000
20	2432902008176640000	-2102132736	2432902008176640000
21	51090942171709440000	OverFlowError	OverFlowError

An entry in binary code is 66 bits long, while the int type in C is 32 bits long (the most significant bit is the sign), and the long long type is 64 bits long. This means that in binary code, discarding the higher bits, we will actually get

int factorial(21) = $(10111000110001000000000000000000)_2$

long long (21) =

$(1100010100000111011111010011011010111000110001000000000000000000)_2$

Taking into account the fact that we are working with sign types and the presented results are written in additive code, the reverse conversion to the decimal numbering system will give the numbers – 1195114496 and –4249290049419214848, respectively, that is, the calculation is correct not from a mathematical point of view, but according to standards and algorithms work with integer types.

In order to increase the accuracy of calculations and obtain correct calculation results, it is often recommended to use floating-point number types instead of integer data types. Sometimes it can give a certain effect, but it is also significantly limited, and the correctness of the obtained results also requires mandatory verification.

We will use the floating point data types presented in Table 1 instead of integer types in the described example of calculating the factorial of a number.

The float32 type ceases to guarantee an exact result already at n = 14:

float32(14!) = $(87178289152)_{10}$

while the correct value 14! = 87178291200. In terms of the IEEE-754 standard, Python calculates again without errors.

After converting to binary code, we have

$(87178291200)_{10} = (1010000100110000111011001010000000000)_2$

float32(87178291200) = 0 (sign) | 10100011 (exponent) | 01000100110000111011001 (mantissa)

that is, we discard "0100000000000" in the lower bits, which is exactly 2048 in the decimal number system. The number 2048 is the error between the actual calculation value of 14! and the result of the calculation is float32(14!).

Starting with n = 23, the type of floating point numbers float64 starts to return an incorrect result. The value of the factorial for n > 25 is incorrect even for the longdouble type, while we will get a value of longdouble(26!) greater than the real value of 26!, which also follows from the peculiarities of the application of the IEEE-754 standard:

longdouble (403291461126605635584000000) =

0 (sign) | 100000001010111 (exponent) |

01001101100110000100100111101010001101111110111010101100100100 10 (mantissa)

In the decimal code, the specified number is converted as 403291461126605635592388608.

3.2. Homer Simpson's numbers

Let's consider another example of manipulating integer formats. The images in Figure 3 are stills from the episode "The Wizard of Evergreen Terrace" [26] and the episode "Treehouse of Horror VI. Homer3 (Homer Cubed)" [27] of the popular animated series The Simpsons.

Mathematical statements relating to Fermat's Great Theorem will be true when checked with a calculator [28]. Let's consider the first example when working with numeric data types from the numpy module.

Of course, the actual numerical values do not match

$$3987^{12} + 4365^{12} = 63976656349698612616236230953154487896987106$$
$$4472^{12} = 63976656348486725806862358322168575784124416$$

Note that an attempt to convert any number to int32, int64 (uint32, uint64) types generates an OverflowError error.

Direct conversion to float32 also generates an OverflowError, so let's try to consider a mathematically identical expression

$$398.7^{12} + 436.5^{12} \,?\, 447.2^{12}$$

Now the result of the operation in Python (398.7**12 + 436.5**12 == 447.2**12) will be True. Indeed, it is fashionable to write within the IEEE-754 standard

Arithmetic expression:	$398.7^{12} + 436.5^{12}$
Correct value:	63976656349698612616236230953154.487896987106
Float32 format:	63976654187609440353486161051648.00
Binary format:	0|11101000|10010011101111111101110
Arithmetic expression:	447.2^{12}
Correct value:	63976656348486725806862358322168.575784124416
Float32 format:	63976654187609440353486161051648.00
Binary format:	0|11101000|10010011101111111101110

That is, the reason lies in discarding the lower bits when converting a number to the float32 format.

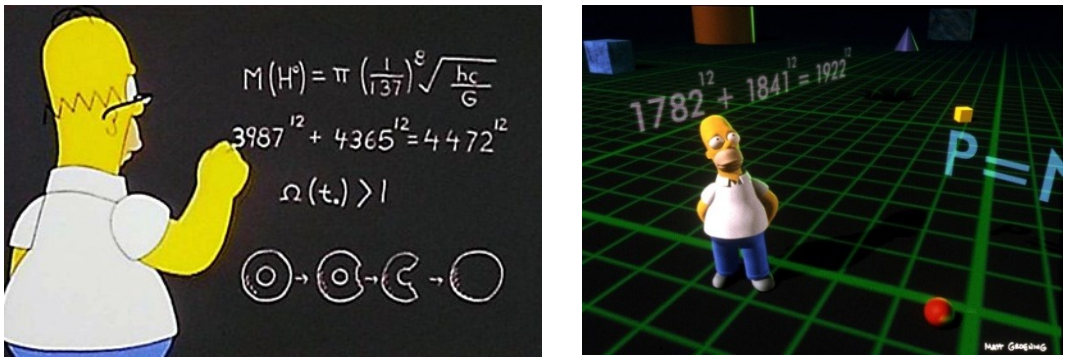

Figure 3: Homer Simpson's example

Let's consider the same problem, but staying within the framework of integer formats.

For example, for n = 5 for the int32 format, more than 1400 triples of natural numbers (A, B, C) can be found, so that when calculating int32(A**5) + int32(B**5) == int32(C**5) would return a boolean value of True.

The maximum value of the parameter C: C = 6208, and as can be seen from the Table 4, several pairs (A, B) can correspond to one value of C. The explanation for this is simple: different large integers (more than 4 bytes) due to the discarding of the high-order bits in the int32 format can have the same value. For triples from the Table 4 (32, B, 6208) we have

int32(6208**5) = 1073741824,
int32(992**5) = int32(5088**5) = 1040187392, int32(32**5) = 33554432.

Similarly for the triples (64, B, 6208) we have

int32(B**5) = 0, int32(64**5) = 1073741824.

Table 4
$C^2 = B^2 + A^2$ (float32)

C	B						A
	992, 5088						32
6208	128,	256,	384,	512,	640,	768,	64
	896,	1024,	1152,	1280,	1408,	1536,	
	1664,	1792,	1920,	2048,	2176,	2304,	
	2432,	2560,	2688,	2816,	2944,	3072,	
	3200,	3328,	3456,	3584,	3712,	3840,	
	3968,	4096,	4224,	4352,	4480,	4608,	
	4736,	4864,	5120,	4992,	5248,	5376,	
	5504,	5632,	5760,	5888,	6016,	6144	

Most of the examples for n = 5 look artificial and any user will immediately understand that the obtained result is incorrect from a mathematical point of view. But you can choose an example when an untrained user can mistakenly perceive the result as correct, for example, implementing C++ programs for the standard type int (4 bytes). From our point of view, this is the example of A = 1504, B = 2960, C = 4496, n = 5. At the same time, it is interesting that the values of the expressions A5 and B5 in the int32 format are generally negative numbers:

int32(1504**5) = -570425344,
int32(2960**5) = -1584398336,
int32(4496**5) = 2140143616.

In the additional code in int32 format, we can write

$(-570425344)_{10} = (11011110000000000000000000000000)_2$
$(-1584398336)_{10} = (10100001100100000000000000000000)_2$

In the binary code, the sum of these numbers will have a length of 33 bits and is equal to

$(-570425344)_{10} + (-1584398336)_{10} = (101111111100100000000000000000000)_2$

Discarding the most significant bit (33 bits), we get the result

$(01111111100100000000000000000000)_2 = (2140143616)_{10}$.

3.3. Another simple examples

Let's move on to consider some typical examples of incorrect behavior of computational algorithms about working with floating point numbers.

Consider the following problem: it is necessary to find such a minimum value of n that

$$\frac{1}{m} + \frac{1}{n} = \frac{1}{2020}, \quad \begin{matrix} m \in N \\ n \in N \end{matrix}, \; n > m$$

This problem is an example of Diophantine equation. In a general sense, the Diophantine equation is an equation in which only integer solutions are allowed.

The program for solving this equation is presented in listing #2.

Listing #2

```
1   import numpy as np
2   def mathExpr (num_format):
3     value = num_format(1/2020)
4     n = 1
5     while True:
6       n1 = num_format(1/n)
7       for m in range(1, n):
8         if n1 + num_format(1/m) == value:
9           return n, m
10      n += 1
11  float_formats = (np.float16, np.float32, np.float64, np.float128)
```

```
12    for form in float_formats:
13        print(f"{form}:", mathExpr (form))
```

```
Result:  <class 'numpy.float16'>:        n = 4039, m = 4035
         <class 'numpy.float32'>:        n = 4041, m = 4039
         <class 'numpy.float64'>:        n = 4545, m = 3636
         <class 'numpy.longdouble'>:     n = 6060, m = 3030
```

As a result, we got four different answers depending on the format of the floating point numbers. At the same time, the pairs (4545, 3636) and (6060, 3030) are the correct solution to the equation.

It is easy to show that n = 4545, if we look for the minimum value of n. That is, when using the float16 and float32 formats, we will get incorrect answers, and in the case of the float128 format, we will get a correct answer from a mathematical point of view, but we will skip over the minimum possible value of n (n = 4545), which satisfies the problem. We can engage the specialized apparatus of computer computation to improve the accuracy of computation. Let's try to increase the accuracy of calculations in Python using a special decimal module [30] by adding a decimal.Decimal element to the float_formats tuple. However, when trying to execute the program, we will get into an infinite loop, and the task of finding an exact match of values can be considered as failed.

We can correct this situation, for example, using the isclose method from the math module. To do this, in listing #2, line 8 must be replaced with the command *if math.isclose(n1+num_format(1/m), value, rel_tol=1e-9)*, and in this case, the float128 and decimal.Decimal format will already have n = 4545, m = 3636. Let's change line 3 in listing #1 to the command value = num_format(1/10), and after starting the program, we will get the result:

<class 'numpy.float16'>: n = 30, m = 15
<class 'numpy.float32'>: n = 35, m = 14
<class 'numpy.float64'>: n = 30, m = 15
<class 'numpy.longdouble'>: ***Infinite loop***
<class decimal.Decimal'>: ***Infinite loop***

Obviously, the answer to the problem is the pair n = 30, m = 15 (the pair n = 35, m = 14 is also a correct result, but the minimum value of the parameter n is still equal to 30). With the double-precision format, the program falls into an infinite loop, that is, the result cannot be calculated.

If we calculate the specified example using the isclose function, then for the case of *rel_tol=1e-7* for all formats of floating point numbers the result will be n = 30, m = 15, and for the value of the *rel_tol* parameter of greater precision (for example , already *rel_tol=1e-8*) – for float16, float64, float128, decimal.Decimal formats the result will be n = 30, m = 15, and for float32 format the result will still be n = 35, m = 14.

We can make an interesting intermediate conclusion: in some examples of computer calculations with floating point numbers, a given lower precision of calculations can give a more correct result than the procedure of increasing the precision This effect is precisely what we observe in the described example: the correct answer is provided by the calculation with the float16 data type (as opposed to float32), the correct result is achieved with the value of the parameter rel_tol=1e-7, and not with the value 0< rel_tol≤ 1e-8.

Finally, we will consider the comparison of calculating the values of the square root of integers using two operations from the math module: sqrt and pow. Despite the simplicity of the problem statement, calculating the square root of a number, the whole root of a number, and especially the inverse square root of a natural number [29] is a relevant task both to correctness of calculations and the time efficiency of algorithms. For example, consider the range of numbers from 1 to 10000, the data type of floating point numbers is standard float (numpy.float64). We will iterate through the values until we find the number value such that math.sqrt(number) != math.pow(number). Table 5 shows the following numbers (for example, less than 10000) depending on the version of Python and the compiler. The table shows only typical examples of the results of computer calculations of the proposed problem. For some

versions (for example, for Python 3.6.5 [GCC 7.3.1], there are no such values under the specified restrictions).

Table 5
Comparison of sqrt and pow functions

Python version	Compiler	Numbers		
3.7.4	GCC 9.2.0			
3.8.10	GCC 9.4.0			
3.9.9	GCC 11.1.0	2921,	3541,	5579,
3.10.6	GCC 11.3.0	7827,	8414,	8415
3.10.12	GCC 11.4.0			
3.11.5	GCC 13.2.1			
3.8.2	GCC 9.3.0	550,	971,	2921,
3.8.5	GCC 9.3.0	3541,	3543,	3884,
3.10.2	Clang 15.0.0	5579,	7827,	8414,
3.11.3	Clang 18.0.0	8415,	8451,	8800

Without going into the technical nuances of the implementation of these algorithms, I will explain the reason for the mismatch of values for the corresponding numbers from Table 5.

math.sqrt(2921) = 54.04627646748664204778833664022386074066162109375
binary(float64) 01000000 01001011 00000101 11101100
01100011 00100101 00110110 11111011

math.pow(2921, 0.5) = 54.04627646748663494236097903922200202941894531250
binary(float64) 01000000 01001011 00000101 11101100
01100011 00100101 00110110 11111010

We see from analysis of the binary representation of calculation results in float64 format that the difference is in the least significant bit of the mantissa. The reason is that the first discarded bit in the representation of the number is 1, and with the calculating math.sqrt(2921) this fact is taken into account. And 1 is added to the least significant bit of the mantissa, but not for calculating math.pow(2921, 0.5). For example, both values are calculated with the increment of the least significant bit taken into account in Python 3.6.5 [GCC 7.3.1].

Conclusions

The work presents the results of the authors' research in the field of methodological foundations of teaching academic disciplines for students of the Computer Science specialty. The authors focused their attention on the topic of accuracy of computer calculations of various problems. This problem is relevant not only when performing engineering and scientific calculations, but also, as shown in the article, when solving the simple training problems. The authors focused on the study of the specified problem using the example of the numpy module, as one of the main software tools for modern scientific calculations.

The authors of the presented study focused on considering fairly simple examples of computer calculations that are completely correct from the point of view of the standards of computer work with numbers and arithmetic operations, the results are incorrect to a human performer. It has been demonstrated that mathematically correct calculations are often impossible to reproduce correctly within the IEEE-754 standard due to the fundamental limitations of the memory allocated to represent a specific number.

Using specific examples, it has been shown how the use of special Python computation features, such as the decimal module, can nevertheless lead to incorrect results. The use of external modules such as mpmath requires further research.

The work also gives paradoxical examples, when less accurate calculations allow you to get a more correct result. Such a statement contradicts the classical practices of calculation methods, and requires a more thorough study of the specifics of computer calculations.

It's important, for the training future specialists in the field of Computer Science during the teaching of subjects related to the accuracy of computer calculations and the fundamental limitations of computer calculations, to present all facts and principles from a single point of view so that students do not there was a false opinion about the arbitrariness of the results of computer calculations and the randomness of the corresponding errors of computer calculations. For a Computer Science professional, it is important not only to understand the basic principles of working with the floating-point number format from the standpoint of the generally accepted IEEE standard, but also to know, even at a basic level, possible modern alternatives, such as, for example, the concept of Posit.

Acknowledgements

All computer calculations and implementation of computer-mathematical models were supported by the computational capacities of the Laboratory of Machine Learning and Intellectual Data Analysis of the Vasyl' Stus Donetsk National University. The laboratory was founded in 2020 by Artem Baiev to conduct academic research and educational work.

References

[1] U. Kulisch, R. Hammer , D. Ratz , and M. Hocks, Numerical Toolbox for Verified Computing I. Basic Numerical Problems. Theory, Algorithms, and Pascal-XSC Programs, Springer Berlin, Heidelberg, 1993. doi:10.1007/978-3-642-78423-1

[2] R. Skeel, "Roundoff Error and the Patriot Missile", SIAM News, 25(4), 1992, 11.

[3] T. Sağlam, "Deadly Round-Off Error: Failure of the Patriot System in Dhahran 1991", Awarded Proseminar and Seminar Papers at the Chair of Software Design and Quality, 1, 2016.

[4] ARIANE 5. Flight 501 Failure, 1996. URL:https://www-users.cse.umn.edu/~arnold/disasters/ariane5rep.html

[5] G. Le Lann, "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective", Proceedings International Conference and Workshop on Engineering of Computer-Based Systems, 1997, 339-346. doi: 10.1109/ECBS.1997.581900

[6] Disasters due to rounding error, 2022. URL:https://web.ma.utexas.edu/users/arbogast/misc/disasters.html

[7] S.C. Chapra, S. Chapra, and D.E. Clough, Applied Numerical Methods with Python for Engineers and Scientists, McGraw Hill, 2022.

[8] P. Dechaumphai, and N. Wansophark, Numerical Methods in Science and Engineering Theories with MATLAB, Mathematica, Fortran, C and Python Programs, Alpha Science International, 2022.

[9] C. Fuhrer, J.E. Solem, and O. Verdier, Scientific Computing with Python: High-performance scientific computing with NumPy, SciPy, and pandas, Packt Publishing, 2021.

[10] R. Johansson, Numerical Python: Scientific Computing and Data Science Applications with Numpy, SciPy and Matplotlib, Apress, 2019. doi:10.1007/978-1-4842-4246-9

[11] "IEEE Standard for Floating-Point Arithmetic" in IEEE Std 754-2019 (Revision of IEEE 754-2008), 2019. doi:10.1109/IEEESTD.2019.8766229

[12] J. Gustafson, The End of Error. Unum Computing, Chapman and Hall/CRC, 2015.

[13] M. Feldman, New Approach Could Sink Floating Point Computation, 2019. URL: https://www.nextplatform.com/2019/07/08/new-approach-could-sink-floating-point-computation/

[14] J. Gustafson and etc., Standard for Posit™ Arithmetic, Posit Working Group, 2022. URL:https://posithub.org/docs/posit_standard-2.pdf

[15] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod etc., "Handbook of Floating-Point Arithmetic", Birkhäuser Basel, 2018. doi:10.1007/978-3-319-76526-6

[16] S. Boldo, C.-P. Jeannerod, G. Melquiond and J.-M. Muller, "Floating-point arithmetic", Acta Numerica, 32 (2023), 203–290. doi:10.1017/S0962492922000101

[17] D.M. Russinoff, Formal Verification of Floating-Point Hardware Design, Springer Cham, 2019. doi:10.1007/978-3-319-95513-1
[18] S. Boldo and G. Melquiond, Computer Arithmetic and Rormal Proofs, ISTE Press - Elsevier, 2017. doi.org:10.1016/C2015-0-01301-6
[19] M.T. Nakao, M. Plum, and Y. Watanabe, "Numerical Verification Methods and Computer-Assisted Proofs for Partial Differential Equations", Springer Singapore, 2019. doi:10.1007/978-981-13-7669-6
[20] O. Vietrov and R. Bilous, "Special methods of increasing the accuracy of computer calculations", 2022 IEEE 3rd KhPI Week on Advanced Technology (KhPIWeek) (2022), 1–5. doi:10.1109/KhPIWeek57572.2022.9916383
[21] O. Vietrov and R. Bilous, "Study of the Convergence of Muller's Sequence Computer Calculations", 2021 IEEE 3rd Ukraine Conference on Electrical and Computer Engineering (UKRCON) (2021), 547–551. doi:10.1109/UKRCON53503.2021.9575546
[22] S.M. Rump, "Algorithms for Verified Inclusions: Theory and Practice", Reliability in Computing: the Role of Interval Methods in Scientific Computing (1988), 109–126. doi:10.15480/882.316
[23] L.G. de la Fraga, "Differential Evolution under FixedPoint Arithmetic and FP16 Numbers", Math. Comput. Appl., 26 (1), 13 (2021). doi:10.3390/mca26010013
[24] Integer Objects, 2024. URL:https://docs.python.org/3/c-api/long.html
[25] Data types, 2023. URL:https://numpy.org/doc/stable/user/basics.types.html
[26] R. Gray, Did Homer Simpson discover the HIGGS BOSON?, 2015. URL: https://www.dailymail.co.uk/sciencetech/article-2975606/Did-Homer-Simpson-discover-HIGGS-BOSON-Maths-1998-episode-predicts-particle-s-mass-14-years-CERN.html
[27] D. Snierson, The Simpsons: A somewhat complete history of 'Homer?' from 'Treehouse of Horror VI', 2018. URL:https://ew.com/tv/2018/10/19/the-simpsons-homer-cubed-treehouse-of-horror-vi/
[28] S. Singh, The Simpsons and Their Mathematical Secrets, Bloomsbury USA, 2013.
[29] C.J. Walczyk, L. Moroz, and J. Cieslinski, "A Modification of the Fast Inverse Square Root Algorithm", Computation, 7(3):41 (2019). doi:10.3390/computation7030041
[30] Decimal fixed point and floating point arithmetic, 2024. URL:https://docs.python.org/3/library/decimal.html