A Need To Specify and Verify Standard Functions

Nikolay V. Shilov
A.P. Ershov Institute of Informatics Systems
Novosibirsk, Russia
shilov@iis.nsk.su

***Abstract* — The problem of validation of standard mathematical functions and libraries is well-recognized by industrial and academic professional community but still is poorly understood by freshmen and inexperienced developers. The paper gives two examples (from author's pedagogical experience) when formal specification and verification of standard functions do help and are needed.**

Keywords* — *mathematical functions; standard libraries; formal specification; formal program verification

I. π IS 4 BACAUSE OF RAND()

A. *What is π?*

How I want a drink, alcoholic of course,

after the heavy lectures involving quantum mechanics.

James Jeans (1877-1946), British Scientist [14]

Mathematical irrational number π is the ratio of a circle's circumference to its diameter; it implies that the quarter of the ration of the area of the circle to the area of a square built on its diameter is also π. This observation leads to Monte Carlo method for computing approximation of π as follows (Figure 1): to draw a segment of a circle in the first quadrant and the square around it, then randomly drop dots in the square; the ratio of dots inside the circle to the total number of dots should be approximately equal π/4. For example, the series of trials depicted in the figure gives π/4≈8/11, i.e. π≈2,(90).

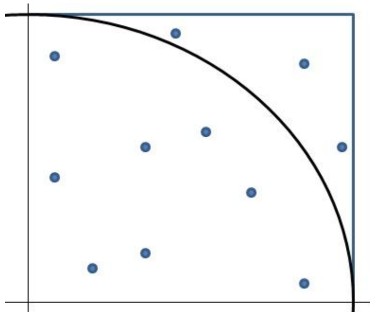

Figure 1: Monte Carlo method to compute π

Of course, the above approximation π≈ 2,(90) is not good[1]. Fortunately, almost everyone remembers much better approximation π≈3.14. Moreover there are many ways to memorize more digits than 3 as above. One way is to memorize a *story* in which the word lengths represent the digits of π: the first word has 3 letters, the second – 1 letter, the third has 4 letters, and so on; in particular, the epigraph of this section is an example of a story to memorize 15 digits of the number.

Some computer languages have a standard function to compute π approximations. For example, the official site `support.office.com` [13] specifies standard PI function and how to use it as shown in Figure 1.

PI function
This article describes the formula syntax and usage of the PI function in Microsoft Excel.

Description

Returns the number 3.14159265358979, the mathematical constant pi, accurate to 15 digits.

Syntax

PI()

The PI function syntax has no arguments

Figure 2: Specification of PI( ) function in MS Excel

B. *Computing π by Monte Carlo*

C-program depicted in Figure 3 implements the above Monte Carlo method to compute an approximation for π. It prescribes to exercise 10 series of 1000000 trials each. This code was developed by a Computer Science instructor to teach first-year students C-loops on base of an interesting and very intuitive algorithm. There were 25 students in the class that used either Code::Blocks 12.11 or Eclipse Kepler IDEs for C/C++ with MinGW environment. Let us refer this program as PiMC (π Monte Carlo) in the sequel.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
int main(void){
srand(time(NULL));
int i, j, r, n = 10;
float pi_val, x, y;
int n_hits, n_trials=1000000;
for(j = 0; j < n; j++){n_hits=0;
        for(i = 0; i<n_trials; i++){
                r = rand()% 10000000;
                x = r/10000000.0;
                r = rand()% 10000000;
                y = r/10000000.0;
                if(x*x + y*y < 1.0) n_hits++;}
        pi_val = 4.0*n_hits/(float)n_trials;
printf("%f \n", pi_val); } return 0;}
```

Figure 3: C-program PiMC to compute π approximation

[1] The Monte Carlo method isn't adaptive and is very slow compared to other methods to compute π.

C. *π equals 4*

Imagine confuse of the instructor when each of 25 students in the class got 10 time value 4.000000 as an approximation for π. (See Figure 4 for a snapshot.)

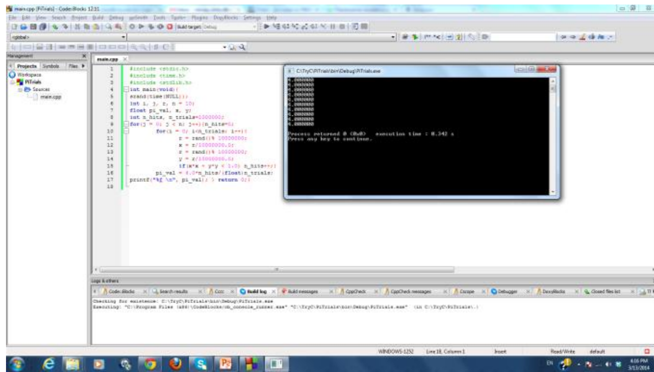

Figure 4: Exercise of program PiMC

But the above outcome of 25 computer experiments was not the last shock for the instructor this day. Because Mathematician that came to run the next class proved that π is really 4. Look at Figure 5 that presents first 3 in a sequence of figures circumscribing a circle with diameter D: each next figure results from the previous one by "cutting corners". The sequence converges to the circle; hence its perimeter converges to πD. But perimeters of all figures in the sequence are constant 4D. Hence π=4.

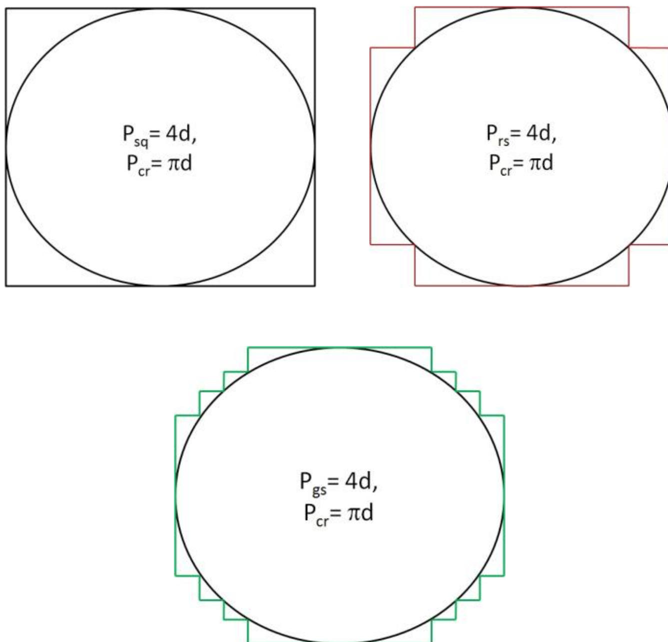

Figure 5: First three figures of a series converging to a circle

So we can summaries: very intuitive Monte Carlo computations and an obvious proof lead us to a paradoxical conclusion that π=4.

D. *Formal Methods as a Rescue*

First let us rule out mathematical "proof" that π=4: presented mathematical arguments don't prove that π=4, but demonstrate that convergence in metrics L_∞ doesn't imply convergence in metrics L_2 [8]: the sequence converges to the circle in metrics L_∞, perimeters of all figures are 4D, but the circumference of the circle is πD≈3.14D.

Next let us try to figure out what is wrong with computer program PiMC. Formal Methods [5] can help this time; in particular let us specify the program in Hoare style by pre- and post-conditions [2].

The pre-condition may be `TRUE` since the program has no input. The post-condition may be `pi_val==4.0` since we know from the program exercise the final value of the variable; but since the real program works with floating point values, maybe it makes sense to use more loose post-condition `3.9<=pi_val<=4.1`. Due to the exercise we may hope that

$$\models[\mathtt{TRUE}]\ \text{PiMC}\ [\mathtt{3.9<=pi_val<=4.1}],$$

i.e. that the total correctness assertion is valid.

But if we try to apply verification methodic from [2] to generate verification conditions and prove the above assertion then we encounter a problem of formal semantics of the function `rand()` in the assignment

```
r = rand()% 10000000;
```

that has 2 instances in the program. The standard rule to generate verification condition for assignment reads

$$\frac{\varphi(x)\rightarrow\psi(t)}{[\varphi(x)]\ \ x=t\ \ [\psi(x)]}\ ;$$

for function `rand()` it leads to

$$\frac{\varphi(x)\rightarrow\psi(\mathtt{rand()})}{[\varphi(x)]\ \ x=\mathtt{rand()}\ \ [\psi(x)]}\ .$$

But, unfortunately, we know too little about properties of this function to prove any non-trivial verification condition! In particular, C reference portal [15] provides just a loose information about this function (Figure 6).

So it is possible to conclude: the cause of wrong π approximation by the program PiMC is use of the standard function `rand()`, its poor specification in the language standard and no verification in MinGW.

rand
C Numerics Pseudo-random number generation
Defined in header <stdlib.h>
int rand();
Returns a pseudo-random integral value between 0 and RAND_MAX (0 and RAND_MAX included).
srand() seeds the pseudo-random number generator used by rand(). If rand() is used before any calls to srand(), rand() behaves as if it was seeded with srand(1). Each time rand() is seeded with srand(), it must produce the same sequence of values.
rand() is not guaranteed to be thread-safe.
Parameters
(none)
Return value
Pseudo-random integral value between 0 and RAND_MAX, inclusive.
Notes
There are no guarantees as to the quality of the random sequence produced. In the past, some implementations of rand() have had serious shortcomings in the randomness, distribution and period of the sequence produced (in one well-known example, the low-order bit simply alternated between 1 and 0 between calls). rand() is not recommended for serious random-number generation needs, like cryptography.
POSIX requires that the period of the pseudo-random number generator used by rand is at least 2^{32}
POSIX offered a thread-safe version of rand called rand_r, which is obsolete in favor of the drand48 family of functions.

Figure 6: Specification of function rand from C reference

II. What is sqrt?

A. Solving Quadratic Equations

An error becomes an error when born as truth.

Stanisław Jerzy Lec (1909-1966),

Polish poet and aphorist [20]

A very popular (but a vulgar for *professional* education) approach to teach standard input/output, floating point data type, sequencing and branching control flow is to program solving of quadratic equations. (Please check [6][17][18] for instance.) Below in Figure 7 one can find a variant of this vulgar code that "solves" quadratic equations in the form $ax^2 + bx + c = 0$.

```
#include <stdio.h>
#include <math.h>
int main(void){
float a, b, c, d, x;
printf("Input coefficients a, b and c and type
'enter' after each:");
scanf("%f%f%f", &a, &b, &c);
d=b*b -4*a*c;
if (d<0) printf("No root(s).");
else {x= (-b + sqrt(d))/(2*a);
      printf("A root is %f.", x);}  return 0;}
```

Figure 7: A vulgar code to "solve" quadratic equation

We put the verb "solves" to quotation marks because non of conventional computers can find root of a simple equation $x^2 - 2 = 0$ (i.e. $\sqrt{2}$) due to irrational nature of the number but finite size all numeric data types in every implementation of C.

Surprisingly, but even C reference [16] says that conventional computers must be able to compute $\sqrt{2}$ (refer to Figure 8).

sqrt, sqrtf, sqrtl
C Numerics Common mathematical functions
Defined in header <math.h>
float sqrtf(float arg);
(1) (since C99)
double sqrt(double arg);
(2)
long double sqrtl(long double arg);
(3) (since C99)
Defined in header <tgmath.h>
#define sqrt(arg)
(4) (since C99)
1-3) Computes square root of arg.
4) Type-generic macro: If arg has type long double, sqrtl is called. Otherwise, if arg has integer type or the type double, sqrt is called. Otherwise, sqrtf is called. If arg is complex or imaginary, then the macro invokes the corresponding complex function (csqrtf, csqrt, csqrtl).
Parameters
arg - floating point value
Return value
If no errors occur, square root of arg ($\sqrt{\text{arg}}$), is returned.
If a domain error occurs, an implementation-defined value is returned (NaN where supported).
If a range error occurs due to underflow, the correct result (after rounding) is returned.
Error handling
Errors are reported as specified in math_errhandling.
Domain error occurs if arg is less than zero.
If the implementation supports IEEE floating-point arithmetic (IEC 60559),
If the argument is less than -0, FE_INVALID is raised and NaN is returned.
If the argument is +∞ or ±0, it is returned, unmodified.
If the argument is NaN, NaN is returned

Figure 8: Specification of `sqrt`-family functions from C reference

But conventional computers can't compute in finite time any irrational number in general and the square root of 2 in particular. It implies that the above specification of the standard function `sqrt` says nonsense and that the function shouldn't be used to solve quadratic equations.

Instead the exact value of irrational square root of 2 a conventional computer can find an approximation of the root with some precision or (it would be better to say) *accuracy*. These approximation and accuracy may be formalized in different ways discussed in the next section.

B. Alternatives for sqrt

For instance, it makes sense to introduce another function with two arguments SQR(Y, E) where Y stays for the argument and E stays for accuracy, that can be formally specified by any (or both) of the following two clauses:

- if Y≥0 and E>0 then SQR(Y, E) differs from $\sqrt{Y}$ less than E, i.e. $|\text{SQR(Y, E)} - \sqrt{\text{Y}}| < E$;
- if Y≥0 and E>0 then $(\text{SQR(Y, E)})^2$ differs from Y less than E, i.e. $|\text{SQR(Y, E)} \times \text{SQR(Y, E)} - \text{Y}| < E$.

Let us fix the first specification for this paper. Then let us select a computation method to compute an approximation. One can select Newton-Raphson Method [7] as a very

intuitive: first guess an initial approximation for the root; then compute arithmetic mean between the guess and the number (whose square root you want to obtain) divided by the initial guess; let this mean to be a new guess for another go-around while the difference between the next and the previous guesses is bigger than the half of the accuracy. The method is easy to implement (see Figure 9).

```
float ab(float X)
{if (X<0) return(-X); else return(X);}

float SQR(float Y, float E)
{float X, D;
X=Y;
do {D=(Y/X-X)/2; X+=D;} while (ab(D)>E/2);
return X;
}
```

Figure 9: Floating-point function to compute an approximation of square root

Both functions in this implementation are easy to specify formally in Hoare style:

- [X is float] ab(X) [returned value is |X|],
- [Y and E are positive floats]

$$\text{SQR(Y,E)}\ [\ |\text{SQR(Y, E)} - \sqrt{\text{Y}}\ | < E\ ].$$

If to prove these specifications, than SQR may be a good alternative to the standard function `sqrt`. Unfortunately, it isn't easy to prove automatically and formally [4] due to several reasons. The major one is axiomatization of computer floating-point arithmetic [1][19].

Even a manual verification of SQR algorithm SQR (assuming precise arithmetic for real numbers) isn't a trivial exercise. Below in Figure 10 one can see a flowchart of (a little bit modified) algorithm of function SQR. (Let us refer the algorithm in the sequel by SQR also.)

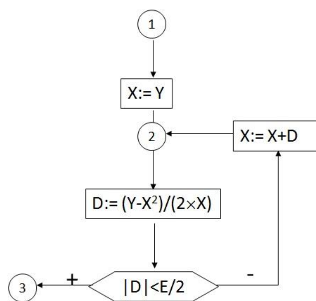

Figure 10: Flowchart of the algorithm implemented in SQR

If to specify the algorithm in line with the function then we need to prove Hoare triple

$$[\text{Y}>0\ \&\ \text{E}>0\ \&\ \text{Y,E}\in\mathbb{R}]\ \text{SQR}\ [\ |\text{SQR(Y, E)} - \sqrt{\text{Y}}\ | < E\ ].$$

In case when Y>1 the corresponding partial correctness assertion can be proved by Floyd method [2] with the following loop invariant (the correctness assertion for the control point 2 in the figure): Y>1 & E>0 & $\sqrt{Y} < X \leq Y$. Halting (termination) of the algorithm can be proved using observation that every time the absolute value of D is twice less (at least) than in the previous iteration.

III. Concluding Remarks

It worth to remark that a need of better specification and validation of standard functions is well-recognized by industrial and academic professional community as well as the problem of conformance of their implementation with the specification [4][9][10][11][12]. Paper [4] addresses the formal verification of some low-level mathematical software for the Intel® Itanium® architecture; in particular it presents details of the verification of a square root algorithm with aid of HOL Light theorem prover. Next two papers [9][10] address formal specification and testing of standard mathematical functions. The last two papers [10][12] present formal specification and verification of some standard memory management and input-output functions.

But a very serious obstacle for formal verification of standard mathematical functions is a need of axiomatization of floating point arithmetic [1][19]. Maybe interval analysis approach [3] and formalization of interval arithmetic may help to tackle the problem for functions like `sqrt` (but not for functions like `rand`).

Unfortunately, the problem (or a pitfall) of poorly specified and verified standard functions and libraries still is poorly understood by freshmen and inexperienced developers. Better education, specification and verification are needed to solve the problem (and avoid the catch of poor libraries).

References

[1] A. Ayad, C. Marché. Multi-prover verification of floating-point programs. Proceedings of Fifth International Joint Conference on Automated Reasoning. Lecture Notes in Artificial Intelligence, 2010, Vol. 6173, p.127-141.

[2] D. Gries, The Science of Programming. Springer-Verlag, 1981.

[3] M.W. Gutowski, Power and beauty of interval methods. arXiv:physics/0302034 [physics.data-an]. Visited October 7, 2015.

[4] J. Harrison, Formal Verification of Square Root Algorithms. Formal Methods in System Design, 2003, Vol.22(2), p.143-153.

[5] C. A. R.. Hoare, The Verifying Compiler: A Grand Challenge for Computing Research. Perspectives of Systems Informatics (PSI'2003), SpringerVerlag, Berlin, LNCS., no. 2890, pp. 1-12, 2003.

[6] S.G. Kochan, Programming in C: A Complete Introduction to the C Programming Language. Exercise #8 at p.162. Sam's Publishing, 2005 (3rd Edition).

[7] S.G. Kochan, Programming in C: A Complete Introduction to the C Programming Language. Functions Calling Functions at p.131. Sam's Publishing, 2005 (3rd Edition).

[8] A.N. Kolmagorov, S.V. Fomin Elements of Funcions Theory and Functional Analysis. Nauka Publishers, 1976 (4th ed., in Russian)

[9] V. Kuliamin, Standardization and Testing of Mathematical Functions Programming and Computer Software, 2007, Vol. 33 (3), p.154-173.

[10] V.V. Kuliamin, Standardization and Testing of Mathematical Functions in floating point numbers. Proceedings of Int. Conf. Perspectives of Systems Informatics PSI-2009. Lecture Notes in Computer Science, 2010, Vol. 5947, p. 257-268.

[11] A.V. Promsky, C Program Verification: Verification Condition Explanation and Standard Library. Automatic Control and Computer Sciences, 2012, Vol. 46, No. 7, p. 394–401.

[12] A.V. Promsky, Experiments on self-applicability in the C-light verification system. Bull. Nov.Comp. Center, Comp. Science, Vol.35, 2013, p.85-99.

[13] Pi Function. https://support.office.com/en-us/article/PI-function-264199d0-a3ba-46b8-975a-c4a04608989b. Visited October 7, 2015.

[14] Pi. Memorizing digits. https://en.wikipedia.org/wiki/Pi#Memorizing_digits. Visited October 7, 2015.

[15] C reference. Rand. http://en.cppreference.com/w/c/numeric/random/rand. Visited October 7, 2015.

[16] C refernce. Sqrt, sqrtf, sqrtl. http://en.cppreference.com/w/c/numeric/math/sqrt. Visited October 7, 2015.

[17] How to make a program that solves the quadratic formula. http://www.youtube.com/watch?v=15NbFrBUdu0. Visited October 7, 2015.

[18] Write a C++ program that solves quadratic equation to find its roots. http://www.cplusplus.com/forum/general/36313/. Visited October 7, 2015.

[19] Hisseo. http://hisseo.saclay.inria.fr/index.html. Visited October 7, 2015.

[20] Stanislaw Jerzy Lec Quotes. http://www.azquotes.com/author/8631-Stanislaw_Jerzy_Lec. Visited October 7, 2015.