On the Need to Specify and Verify Standard Functions

N.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*

1 π is 4

1.1 What is π?

How I want a drink, alcoholic of course,
after the heavy lectures involving quantum mechanics.
James Jeans (1877-1946), British Scientist [15]

The mathematical irrational number π is the ratio of a circle's circumference to its diameter D; it is also well-known mathematical fact that the area of a circle is $(\pi \times D^2)/4$, i.e. it is the $\pi/4$ of the area of the square built on the circle's diameter. This observation leads to Monte Carlo method[1] for computing approximation of π as follows (Fig. 1): to draw a segment of a circle in the first quadrant and the square around it, then randomly place dots in the square; the ratio of dots inside the circle to the total number of dots should be approximately equal $\pi/4$. For example, the series of trials depicted in the figure gives $\pi/4 \approx \frac{8}{11}$, i.e. $\pi \approx 2.(90)$.

Of course, the above approximation $\pi \approx 2.(90)$ is not good. Fortunately, almost everyone remembers much better approximation $\pi \approx 3.14$. 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.

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

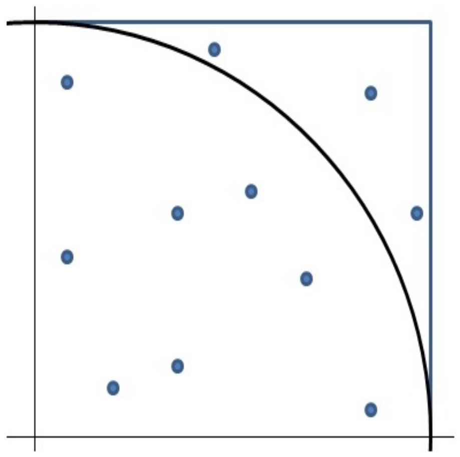

Fig. 1. Monte Carlo method to compute π

Some computer languages have a standard function to compute π approximations. For example, the official site `support.office.com` specifies standard `PI` function and how to use it as follows [16]:

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

1.2 π by Monte Carlo

An error becomes an error when born as truth.
Stanisław Jerzy Lec (1909-1966),
Polish poet and aphorist [17]

The C-program depicted in Fig. 2 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;}
```

Fig. 2. C-program PiMC to compute π approximation

Imagine the confusion of the instructor when each of 25 students in the class got 10 times value 4.000000 as an approximation for π! But it wasn't the last shock for the instructor this day. Because Mathematician that came to run the next class proved that is really 4. Look at Fig. 3 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 $\pi \times D$. But perimeters of all figures in the sequence is constant $4D$. Hence $\pi = 4$.

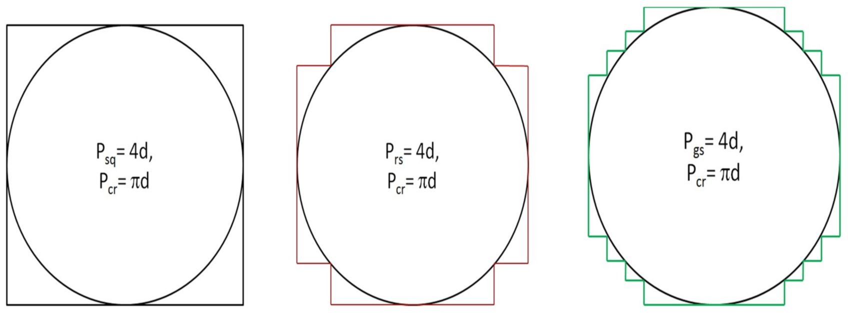

Fig. 3. First three figures of a series converging to a circle

So we can summaries: a very intuitive Monte Carlo computational experiment repeated independently 25 times and an obvious proof lead us to a paradoxical conclusion that π is 4.

1.3 Formal Methods as a Rescue

First let us rule out mathematical "proof" that $\pi = 4$: presented mathematical arguments don't prove that $\pi = 4$, but instead demonstrate that convergence in metrics $\mathbf{L}_\infty$ doesn't imply convergence in metrics $\mathbf{L}_\infty$ [8]: the sequence converges to the circle in metrics $\mathbf{L}_\infty$, perimeters of all figures are $4D$, but the circumference of the circle is $\pi \times D \approx 3.14D$.

Next let us try to figure out what is wrong with computer program PiMC with aid of Formal Methods [5]; in particular let us try to specify the program in the classical 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. Due to the exercise we may formulate the following hypothesis

$$\models \texttt{[TRUE] PiMC [pi\_val==4.0]}, \tag{1}$$

i.e. that the total correctness assertion `[TRUE] PiMC [pi_val==4.0]` is valid.

But if we try to apply the classical 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

$$\texttt{r=rand()\%10000000;} \tag{2}$$

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

$$\frac{\phi(x) \to \psi(t)}{[\phi(x)]\ x = t\ [\psi(x)]};$$

for function `rand()` it leads to the following rule

$$\frac{\phi(x) \to \psi(\texttt{rand()})}{[\phi(x)]\ \texttt{x=rand()}\ [\psi(x)]}.$$

But, unfortunately, we know too little about properties of this function to prove any non-trivial verification condition! In particular, from intuitive *very informal* understanding of a random-value generator, `rand()` should generate with equal probability all values from some range[2] *Range*; it implies that the premise $\phi(x) \to \psi(\texttt{rand()})$ in the rule for `rand()` is equivalent to the formula $\phi(x) \to \forall x \in \mathit{Range}.\ \psi(x)$. This verification condition generation rule looks very special (not to say suspicious). Due to this reason, Hoare-style verification for

[2] For C-language it is known (please refer the next subsection 1.4) that this *Range* is an integer interval [0..RAND_MAX].

probabilistic programs have got a little attention in the past [4] and still is a research topic [14].

Nevertheless after the above discussion one may conclude that the cause of wrong π approximation by the program PiMC is the use of the assignments (2) and (maybe) the use of the standard function `rand()`, its poor specification in the language standard and no verification in MinGW.

1.4 How to Fix It

Remark. *This subsection is due to reviewer's comments for the initial version of the paper. The author is very much oblige to the anonymous reviewer and would like to thanks him/her for the very valuable addendum.*

C-language reference portal at `en.cppreference.com/w/c` provides the following loose specification for the function `rand()` [18]:

C Numerics Pseudo-random number generation
rand
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.

Of course this specification isn't very formal and precise and, hence, can't be helpful to prove every formal property of every program that uses function `rand()`. Nevertheless this specification contains enough information to detect what is wrong with the assignment (2) in the program PiMC.

Recall that the function `rand()` returns an integer in the range from 0 to `RAND_MAX` inclusively. In many conventional implementation of the C language (including MinGW) `RAND_MAX` is $2^{15} - 1 = 32767 < 1000000$. It implies that the assignment (2) is simply equivalent to the assignment

```
r = rand();
```

Since the value of `RAND_MAX` in our experiment is just 32767, the normalizing assignments

```
x = r/10000000.0;
.................
y = r/10000000.0;
```

result with values of `x` and `y` in the range $[0, 0.032767]$; hence, the condition `x*x + y*y < 1.0` in the following on if-operator is always true and the program PiMC always increments the value of the variable `n_hits` that counts the number of "randomly" dropped points that have fallen inside the segment of the circle. It implies that every time after termination of the internal for-loop `for (i = 0; i<n_trials; i++)` values of the variables `n_hits` and `n_trials` are always equal, and consequently the final value of `pi_val` is always exactly 4. We can consider the above discussion as a refutation of for the hypothesis 1 but as an informal proof for

$\models$`[rand()`$\in$`[0..RAND_MAX] & RAND_MAX==32767] PiMC [pi_val==4.0]`.

The above consideration and discussion lead to the idea how to fix program PiMC. The body of the internal for-loop should be replaced for instance by the following code:

```
x = rand()/(float)RAND_MAX;
y = rand()/(float)RAND_MAX;
if(x*x + y*y < 1.0) n_hits++;
```

Let us denote the fixed code by FixPi. Then one can run the program and get very reasonable approximations for the irrational number π. For example, the anonymous reviewer has got values 3.140932, 3.141289, ... 3.141315 in a row. The mean value is 3.141353. Due to the exercise we may formulate a new hypothesis

$\models$`[RAND_SPEC & RAND_MAX==32767] FixMC [3.140<=pi_val<=3.142]`,

where `RANS_SPEC` stays for a *formal specification* of `rand()`. But we have to say that we still don't know how a *loose specification* of `rand()` from C-language reference portal can help to prove or refute the above total correctness assertion.

```
#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;}
```

Fig. 4. A vulgar solver for quadratic equations

2 What is SQRT?

2.1 Solving Quadratic Equations

A very popular (but 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, 19, 20] for instance.) In Fig. 4 one can find a variant of a vulgar solver for quadratic equations in the form $ax^2 + bx + c = 0$. We call this solver "vulgar" because none of conventional computers can solve (in pure mathematical sense, i.e. to find a root) a simple equation $x^2 - 2 = 0$ (i.e. to compute $\sqrt{2}$) due to irrational nature of the root but finite size all numeric data types in every implementation of language C.

To clarify an ambiguity with conventional computer ability to solve quadratic equations, let us check what is said at C reference portal at `en.cppreference.com/w/c` regarding "square root function" `sqrt` [21]:

C Numerics Common mathematical functions
sqrt, sqrtf, sqrtl
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 ($\sqrt{arg}$), 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 $+\infty$ or ± 0, it is returned, unmodified.
• If the argument is NaN, NaN is returned
Notes
sqrt is required by the IEEE standard be exact.
The only other operations required to be exact are the arithmetic operators
and the function fma. After rounding to the return type (using default rounding mode),
the result of sqrt is indistinguishable from the infinitely precise result.
In other words, the error is less than 0.5 ulp.
Other functions, including pow, are not so constrained.

One can see an ambiguity in the specification. According to the above citation, the specification first says that `sqrt(2)` must be $\sqrt{2}$, but then (in the Notes) that the error of `sqrt(2)` must less than 0.5 of ulp — the unit in the least precision (that is type and platform dependable), because the function *required by the IEEE standard be exact.* Of course we have to rule out the first option (that `sqrt()` is $\sqrt{\ }$) and examine in more details the second one (that `sqrt()` computes $\sqrt{\ }$ with error less than 0.5ulp).

The standards that are mentioned in the specification are IEEE 754-2008 Standard for Floating-Point Arithmetic and the international standard ISO/IEC 60559:2011 [22] (that is identical to IEEE 754-2008). Section 9 of the standard recommends fifty operations, that language standards should define (but all these operations are optional, not required in order to conform to the standard). But some operations (including `sqrt()` as a special case of the function $(\ )^{1/n}$ for $n = 2$), if being implemented, must round correctly (i.e. with an error less than 0.5ulp).

Another very critical problem with the specification and ISO/IEC/IEEE standards above is absence (in the specification and standards) of description of any validation procedure to check/prove that an implementation conforms to the specification/standard.

2.2 An Alternative for `sqrt`

Instead requiring from `sqrt` to compute the exact irrational square roots or type- and/or platform-dependent approximate values of square roots, it make

```
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;}
```

Fig. 5. Floating-point function to compute an approximation of square root

sense to specify in the language another "standard" *generic* function (say `SQR( , )`) for all numeric types with returen value and two parameters of this numeric type, where the first parameter is for passing the argument value $Y \geq 0$ and the second is for passing the accuracy value $E > 0$, to compute $\sqrt{Y}$ with accuracy E. Let us use the following notation for the suggested language function `SQR`: SQR is a mathematical function of two real arguments that is computed by `SQR`, i.e. for every non-negative real number $Y \geq 0$ and positive real number $E > 0$, `SQR(`Y`,`E`)` returns $SQR(Y, E)$. Properties of this function SQR can be formally specified by any (or both) of the following two clauses:

– if $Y \geq 0$ and $E > 0$ then $SQR(Y, E)$ differs from $\sqrt{Y}$ not more than E, i.e. $|SQR(Y) - \sqrt{Y}| \leq E$;
– if $Y \geq 0$ and $E > 0$ then $(SQR(Y, E))^2$ differs from Y not more than E, i.e. $|(SQR(Y))^2 - Y| \leq E$.

It makes sense to fix the first formal specification for better compatibility with the exact standard function, since in this case we can define the standard function `sqrt` via `SQR` as follows:

```
float sqrt(float Y)
{return((float)SQR(Y, default(float)/2.0);}
```

where `default` is another new type-related feature (as `sizeof`) that returns the unit in the least precision of the type.

One may select any reasonable and feasible computation method to approximate $\sqrt{\ }$. For example it can be a very intuitive Newton-Raphson Method [7]: 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 the new guess for another go-around while the difference between the next and the previous guess is bigger than the half of the accuracy. (Please refer Fig. 5 for a sample implementation of the function for `float` data type.)

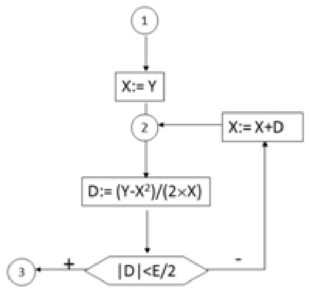

Fig. 6. Flowchart of the algorithm *SQR* implemented in function `SQR`

Both functions in Fig. 5 are easy to specify[3] formally in Hoare style:

$$\texttt{[}X\texttt{ is float] ab(}X\texttt{) [returned value is }|X|\texttt{],}$$

$$\texttt{[}Y \geq 0\texttt{ and }E > 0\texttt{ are floats]} \quad \texttt{SQR(}Y\texttt{,}E\texttt{) [}|\texttt{returned value}-\sqrt{Y}| \leq E\texttt{].} \tag{3}$$

If to prove these specifications, than `SQR` may be a good alternative to the standard function `sqrt`. Unfortunately, it isn't easy to prove these specifications automatically and formally due to several reasons. The major one is axiomatization of computer floating-point arithmetic [1, 11, 23]. Even a manual pen-and-paper verification of algorithm *SQR* (assuming precise arithmetic for real numbers) isn't a trivial exercise that we solve in the next subsection.

2.3 Toward Formal Verification of *SQR*-algorithm

In Fig. 6 one can see a flowchart of (a little bit modified) algorithm of the function `SQR` in Fig. 5. Let us refer the algorithm as *SQR* in the sequel. If to specify the algorithm in the same way as the function then we need to prove the following "relaxation" of the second triple in (3):

$$[Y, E \in \mathbb{R} \ \& \ Y \geq 0 \ \& \ E > 0] \ SQR \ [|x - \sqrt{Y}| \leq E] \tag{4}$$

To prove this assertion, let us consider three disjoint cases for the range of the initial value of the variable Y: $0 \leq Y < 1$, $Y = 1$ and $Y > 1$:

$$[Y, E \in \mathbb{R} \ \& \ 0 \leq Y < 1 \ \& \ E > 0] \ SQR \ [|x - \sqrt{Y}| \leq E], \tag{5}$$

$$[Y, E \in \mathbb{R} \ \& \ Y = 1 \ \& \ E > 0] \ SQR \ [|x - \sqrt{Y}| \leq E], \tag{6}$$

$$[Y, E \in \mathbb{R} \ \& \ Y > 1 \ \& \ E > 0] \ SQR \ [|x - \sqrt{Y}| \leq E]. \tag{7}$$

Since the second case (6) is trivial, two other cases — (5) and (7) — are very similar, let us consider the last one only, i.e. the assertion (7).

[3] Remark that the specification is incomplete since it doesn't specify exceptional situations (e.g. $Y < 0$ and/or $E \leq 0$)

Partial Correctness. Let us employ Floyd method [2] for pen-and-paper proof of partial correctness. Let us select control points 1, 2 and 3 as is depicted in Fig. 6 to cut the flowchart on three loop-free pathes

path (1..2) from starting point 1 to point 2;
path (2+3) from point 2 to final point 3 via positive branch after choice;
path (2–2) from point 2 to the same point 2 via negative branch after choice.

Let us consider all these paths one by one using the following annotations for the control points:

1. $Y > 1 \ \& \ E > 0$ (i.e. the pre-condition);
2. $Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y$ (the loop invariant);
3. $|x - \sqrt{Y}| \leq E$ (i.e. the post-condition).

The first path (1..2) is easy to verify:

$$\frac{(Y > 1 \ \& \ E > 0) \to (Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < Y \leq Y)}{\{Y > 1 \ \& \ E > 0\} \ X := Y \ \{Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y\}}.$$

The second path (2+3) isn't so easy. Let us introduce test program construct $\phi?$ as a short-hand for *if ϕ then stop else abort*. Then verification of the path (after some simplification) follows:

$$\frac{\dfrac{(Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y \ \& \ |\frac{Y-X^2}{2X}| < E/2) \ \to \ |X - \sqrt{Y}| < E}{\{Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y\} \ D := \frac{Y-X^2}{2X} \ \{|D| < E/2 \ \to \ |X - \sqrt{Y}| < E\}}}{\{Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y\} \ D := \frac{Y-X^2}{2X} \ ; \ |D| < E/2? \ \{|X - \sqrt{Y}| < E\}}$$

The premise

$$(Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y \ \& \ |\frac{Y - X^2}{2X}| < E/2) \ \to \ |X - \sqrt{Y}| < E$$

is valid since in this case we have

$$|X - \sqrt{Y}| = \Big(\frac{|X - \sqrt{Y}| \ (X + \sqrt{Y})}{2X}\Big) \times \Big(\frac{2X}{X + \sqrt{Y}}\Big) < \frac{E}{2} \times \frac{2}{1 + \frac{\sqrt{Y}}{X}} < E.$$

Proof (also after some simplification) of the third path (2–2) follows:

$$\frac{\dfrac{(Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y \ \& \ |\frac{Y-X^2}{2X}| \geq E/2) \ \to \ (Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < \frac{Y+X^2}{2X} \leq Y)}{\begin{array}{l}\{Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y\} \ D := \frac{Y-X^2}{2X} \\ \qquad\qquad \{|D| \geq E/2 \ \to \ (Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X + D \leq Y)\}\end{array}}}{\begin{array}{l}\{Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y\} \ D := \frac{Y-X^2}{2X} \ ; \\ \qquad\qquad |D| \geq E/2? \ ; \ X := X + D \ \{Y > 1 \ \& \ E > 0 \ \& \ \sqrt{Y} < X \leq Y\}\end{array}}$$

Hint to prove the premise of this derivation: use that $D < 0$ and that geometric mean isn't grater than arithmetic one $\sqrt{Y} < \frac{Y+X^2}{2X}$.

Termination. Let us observe that the loop invariant $Y > 1$ & $E > 0$ & $\sqrt{Y} < X \leq Y$ implies that every loop iteration reduces the absolute value of D twice at least.

Let us fix some $y > 1$ as the initial value of the variable Y, $\varepsilon > 0$ as the initial value of the variable E, let x_1, x_2, ... x_n, $x_{(n+1)}$, ... be values of the variable X immediately *before* 1^{st}, 2^{nd}, ... n-th, $(n+1)$-th, etc., iteration of the loop for this fixed initial value y of Y, and let d_1, d_2, ... d_n, $d_{(n+1)}$, ... be values of the variable D immediately *after* 1^{st}, 2^{nd}, ... n-th, $(n+1)$-th, etc., iteration of the loop (also for the same fixed initial value y of Y). In particular, $x_1 = y$ and $d_n = \frac{y-x_n^2}{2x_n}$, $x_{(n+1)} = x_n + d_n$ for all $n.0$.

Let us express $d_{(n+1)}$ in terms of d_n:

$$d_{(n+1)} = \frac{y-x_{(n+1)}^2}{2x_{(n+1)}} = \frac{y-(x_n+d_n)^2}{2(x_n+d_n)} = \frac{y-(\frac{y+x_n^2}{2x_n})^2}{2\frac{y+x_n^2}{2x_n}} = \\ = -\frac{(y-x_n^2)^2 x_n}{4x_n^2(y+x_n^2)} = -\frac{d_n^2 x_n}{y+x_n^2} = -\frac{d_n^2}{2x_{(n+1)}}.$$

Remark that all values d_1, d_2, ... d_n, $d_{(n+1)}$, ... are negative due to loop invariant. Hence

$$\frac{|d_{(n+1)}|}{|d_n|} = \frac{d_{(n+1)}}{d_n} = -\frac{d_n}{2x_{(n+1)}} = \frac{1}{2} \times \frac{x_n^2 - y}{x_n^2 + y} < \frac{1}{2}.$$

It implies that $|d_{(n+1)}| < \frac{y}{2^n}$, i.e. the algorithm terminates after (at most) $\log_2 \frac{y}{\varepsilon}$ iterations of the loop.

3 Concluding remarks

Let us start these concluding remarks with quotation from the abstract of the paper [11], because it correlates with the present paper very well:

> *Current critical systems commonly use a lot of floating-point computations, and thus the testing or static analysis of programs containing floating-point operators has become a priority. However, correctly defining the semantics of common implementations of floating-point is tricky, because semantics may change with many factors beyond source-code level, such as choices made by compilers. We here give concrete examples of problems that can appear and solutions to implement in analysis software.*

The major difference between [11] and the present paper is the concern: cited paper addresses problems with floating point value representation and arithmetics, but the present paper — the problems with the standard functions specification and implementation.

It is worth to remark that a need of better specification and validation of the standard functions is recognized (in principle) by industrial and academic professional community as well as the problem of conformance of their implementation with the specification [3, 9, 10, 12, 13].

Paper [3] 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 cited papers [12, 13] 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, 23]. Maybe interval analysis approach and formalization of interval arithmetic may help to tackle the problem for functions like `sqrt` but not for functions like `rand`; for functions like `rand` maybe we need to develop and employ some special methods for probabilistic programs [4, 14].

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. Ayad A., Marché C. Multi-prover verification of floating-point programs // Lecture Notes in Artificial Intelligence. 2010. Vol. 6173. P.127–141.
2. Gries D. The Science of Programming. Springer-Verlag, 1981.
3. Harrison J. Formal Verification of Square Root Algorithms // Formal Methods in System Design. 2003. Vol. 22, n. 2. P.143–153.
4. Den Hartog J.I. and de Vink E.P. Verifying probabilistic programs using a hoare like logic // International journal of foundations of computer science. 2002. Vol. 13, n.3. P.315–340.
5. Hoare C.A.R. The Verifying Compiler: A Grand Challenge for Computing Research // Lecture Notes in Computer Science. 2003. Vol. 2890. P. 1–12.
   Gutowski M.W. Power and beauty of interval methods. arXiv:physics/0302034 [physics.data-an]. urlhttp://arxiv.org/pdf/physics/0302034.pdf (visited January 19, 2016).
6. Kochan S.G. Programming in C: A Complete Introduction to the C Programming Language. Exercise ♯8 at p.162. Sam's Publishing, 2005 (3rd Edition).
7. Kochan S.G. Programming in C: A Complete Introduction to the C Programming Language. Functions Calling Functions at p.131. Sam's Publishing, 2005 (3rd Edition).
8. Kolmagorov A.N., Fomin S.V. Elements of Funcions Theory and Functional Analysis. Nauka Publishers, 1976 (4th ed., in Russian).
9. Kuliamin V. Standardization and Testing of Mathematical Functions // Programming and Computer Software. 2007. Vol. 33, n. 3. P. 154–173.
10. Kuliamin V.V. Standardization and Testing of Mathematical Functions in floating point numbers // Lecture Notes in Computer Science. 2010. Vol. 5947. P. 257–268.
11. Monniaux D. The pitfalls of verifying floating-point computations // ACM Transactions on Programming Languages and Systems. 2008. Vol. 30, n. 3. P.1–41.

12. Promsky A.V. C Program Verification: Verification Condition Explanation and Standard Library // Automatic Control and Computer Sciences. 2012. Vol. 46, n. 7. P. 394–401.
13. Promsky A.V. Experiments on self-applicability in the C-light verification system // Bull. Nov.Comp. Center, Comp. Science Series. 2013. Vol.35. P. 85–99.
14. Rand R. and Zdancewic S. VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs // Electronic Notes in Theoretical Computer Science. 2015. Vol. 319. P.351–367.
15. Pi. Memorizing digits. `https://en.wikipedia.org/wiki/Pi#Memorizing_digits` (visited January 19, 2016).
16. Pi Function. `https://support.office.com/en-us/article/PI-function-264199d0-a3ba-46b8-975a-c4a04608989b` (visited January 19, 2016).
17. Stanislaw Jerzy Lec Quotes. `http://www.azquotes.com/author/8631-Stanislaw_Jerzy_Lec` (visited January 19, 2016).
18. C reference. `Rand.http://en.cppreference.com/w/c/numeric/random/rand` (visited January 19, 2016).
19. How to make a program that solves the quadratic formula. `http://www.youtube.com/watch?v=15NbFrBUdu0` (visited January 19, 2016).
20. Write a C++ program that solves quadratic equation to find its roots. `http://www.cplusplus.com/forum/general/36313/` (visited January 19, 2016).
21. C refernce. Sqrt, sqrtf, sqrtl. `http://en.cppreference.com/w/c/numeric/math/sqrt` (visited January 19, 2016).
22. ISO/IEC/IEEE 60559:2011. Information technology -- Microprocessor Systems -- Floating-Point arithmetic `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57469` (visited January 19, 2016).
23. Hisseo. `http://hisseo.saclay.inria.fr/index.html` (visited January 19, 2016).