Platform-independent model of fix-point arithmetic for verification of the standard mathematical functions

Nikolay V. Shilov (Innopolis University)

Dmitry A. Kondratyev (Institute of Informatics Systems, Novosibirsk)

Boris L. Faifel (Technical University of Saratov)
Russia

Platform-independent approach

Part 0

In brief

- Our research project is aimed onto a development of an incremental combined approach to the specification and verification of the standard mathematical functions.
- *Platform independence* means that we attempt to design a relatively simple axiomatization of the computer arithmetic in terms of real, rational, and integer arithmetic (i.e., the fields $\boldsymbol{R}$ and $\boldsymbol{Q}$ of real and rational numbers, the ring $\boldsymbol{Z}$ of integers) but don't specify neither base of the computer arithmetic, nor a format of numbers' representation.

More details

- *Incrementality* means that we start with the most straightforward specification of the simplest easy to verify algorithm in real numbers and finish with a realistic specification and a verification of an algorithm in computer arithmetic.
- We call our approach *combined* because we start with a manual (pen-and-paper) verification of some selected algorithm in real numbers, then use these algorithm and verification as a draft and proof-outlines for the algorithm in computer arithmetic and its manual verification, and finish with a computer-aided validation of our manual proofs with some proof-assistant system (to avoid appeals to \obviousness" that are very common in human-carried proofs).

What’ wrong with specification and computing standard Math functions

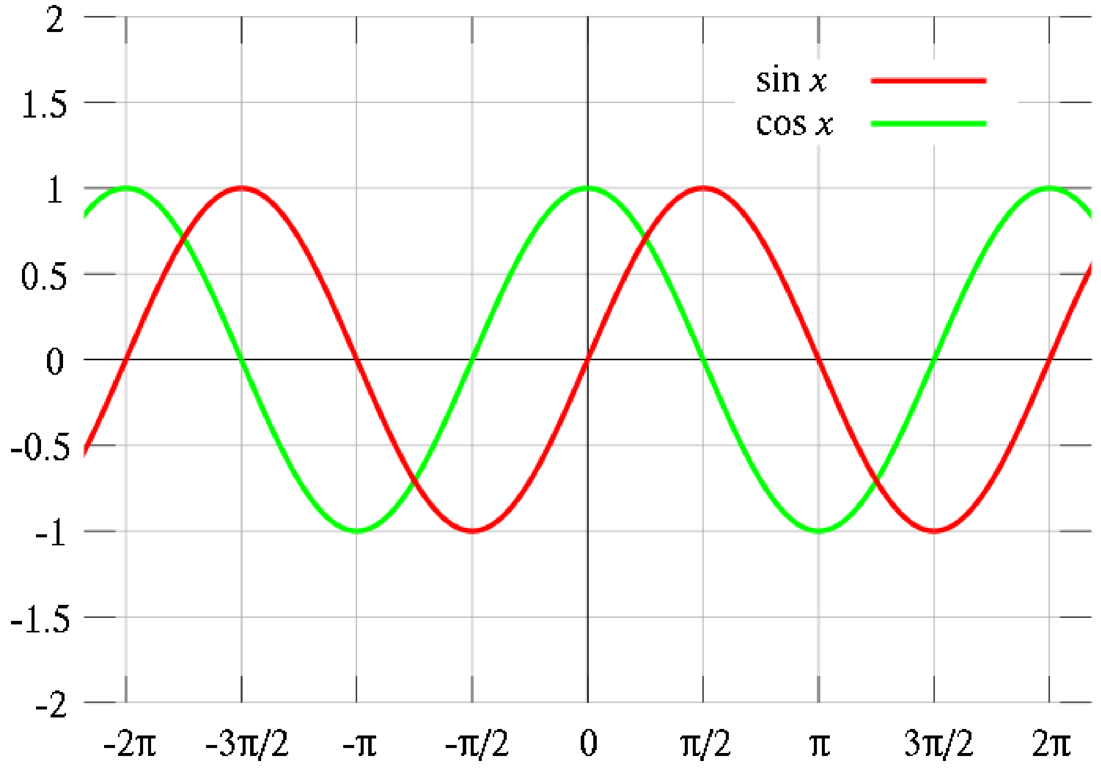
Part I

Sine and cosine

Properties of sin and cos :

- $|\sin x| \leq 1$ and $|\cos x| \leq 1$,
- $\cos^2 x + \sin^2 x = 1$,
- $\sin(-x) = -\sin x$ and $\cos(-x) = \cos x$,
- …

for all $x \in \boldsymbol{R}$.



How to compute sine and cosine

- The following Maclaurin/Taylor expansion for (real) sine, cosine and exponent are valid:
    - $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} + \cdots - \frac{x^{4n+2}}{(4n+2)!} + \frac{x^{4n+4}}{(4n+4)!} + \ldots$
    - $\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + \frac{x^{4n+1}}{(4n+1)!} - \frac{x^{4n+3}}{(4n+3)!} + \ldots$
    - $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots + \frac{x^n}{n!} + \ldots$

How to compute sine and cosine – cont.

- Expansions for sine and cosine converge absolutely on the Euclidian plane and are alternating series, i.e. the absolute value of sum of any “tail” is always not greater of the absolute value of the last used term:
  - $\left|\frac{x^{4n-2}}{(4n-2)!}\right| \geq \left|-\frac{x^{4n+2}}{(4n+2)!} + \frac{x^{4n+4}}{(4n+4)!} + \ \ldots\right|,$
  - $\left|\frac{x^{4n+1}}{(4n+1)!}\right| \geq \left|\frac{x^{4n+1}}{(4n+1)!} - \frac{x^{4n+3}}{(4n+3)!} + \ \ldots\right|.$

Computing in C

```
float Cos(float x, float eps)
{
      float s,a,n;
      s=a=n=1;
      while (1)
      {
         a=-a*x*x/(n*(n+1));
         s=s+a;
         if (fabs(a) <= eps) break;
         n+=2;
      }
      return s;
}
```

```
int main()
{
    float x,c,C,d;
    for (x=0; x<30; x=x+0.05)
    {
        c=cos(x);
        C=Cos(x,0.000001);
        d=fabs(c-C);
        printf("%e  %e  %e
  %e\n",x,c,C,d);
   }
   return 0;
}
```

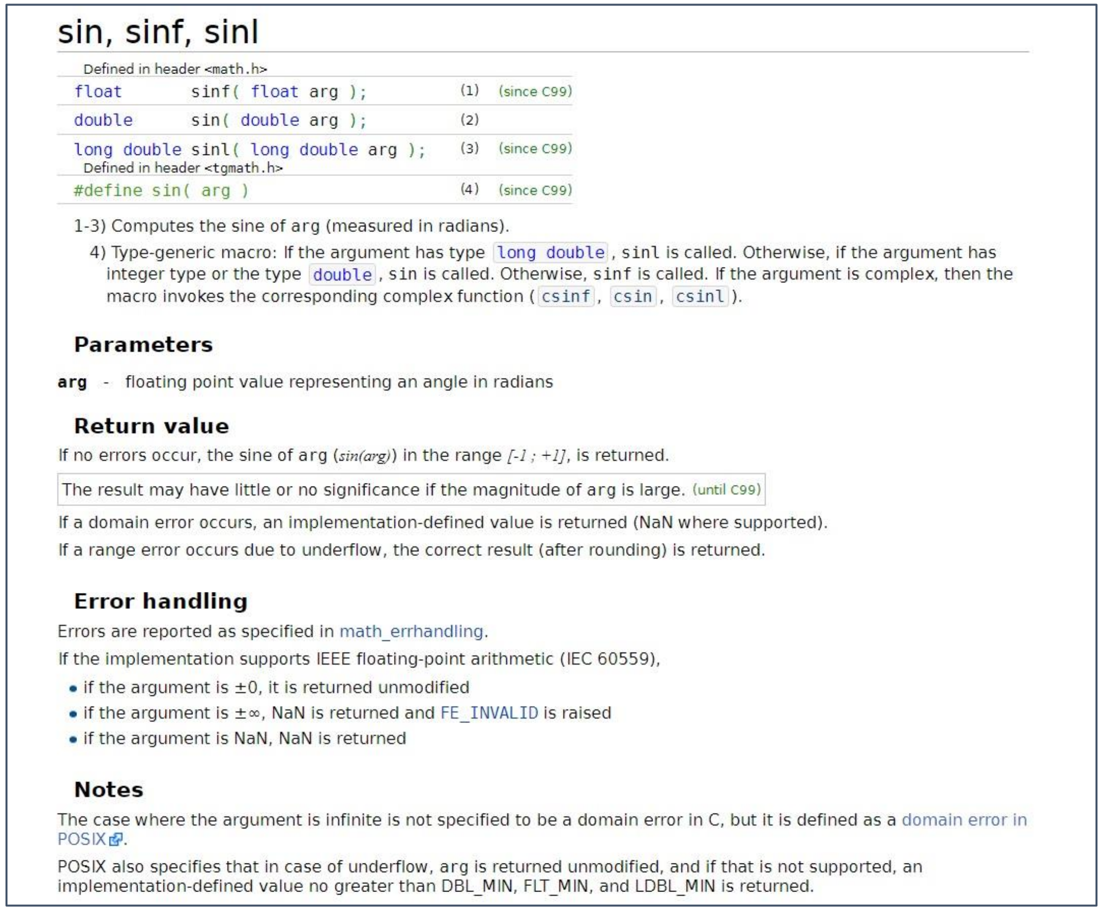
Catastrophe!!!

x	cos(x)	Cos(x)	Расхождение
0.000e+000	1.000000e+000	1.000000e+000	0.000000e+000
5.000e-002	9.987503e-001	9.987502e-001	1.000000e-008
1.000e-001	9.950042e-001	9.950042e-001	0.000000e+000
1.500e-001	9.887711e-001	9.887711e-001	0.000000e+000
2.000e-001	9.800666e-001	9.800666e-001	0.000000e+000
1.850e+001	9.395249e-001	1.144768e+000	2.052426e-001
1.900e+001	9.887046e-001	9.913036e-001	2.599001e-003
1.950e+001	7.958150e-001	6.106047e-001	1.852103e-001
2.900e+001	-7.480575e-001	1.315880e+004	1.315954e+004
2.950e+001	-3.383192e-001	3.822034e+003	3.822372e+003
3.000e+001	1.542515e-001	-2.368533e+003	2.368688e+003

C reference says…

(https://en.cppreference.com/w/c/numeric/math/sin)

- Return value
  - If no errors occur, the sine of arg ($\sin(arg)$) in the range $[-1; +1]$, is returned.
  - *The result may have little or no significance if the magnitude of arg is large.* (until C99)
  - 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.



sin, sinf, sinl

Defined in header <math.h>

```
float       sinf( float arg );                (1)  (since C99)
double      sin( double arg );                (2)
long double sinl( long double arg );          (3)  (since C99)
```

Defined in header <tgmath.h>

```
#define sin( arg )                            (4)  (since C99)
```

1-3) Computes the sine of arg (measured in radians).

4) Type-generic macro: If the argument has type `long double`, `sinl` is called. Otherwise, if the argument has integer type or the type `double`, `sin` is called. Otherwise, `sinf` is called. If the argument is complex, then the macro invokes the corresponding complex function (`csinf`, `csin`, `csinl`).

Parameters

arg	-	floating point value representing an angle in radians

Return value

If no errors occur, the sine of arg (*sin(arg)*) in the range *[-1 ; +1]*, is returned.

The result may have little or no significance if the magnitude of arg is large. (until C99)

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.

If the implementation supports IEEE floating-point arithmetic (IEC 60559),

- if the argument is ±0, it is returned unmodified
- if the argument is ±∞, NaN is returned and FE_INVALID is raised
- if the argument is NaN, NaN is returned

Notes

The case where the argument is infinite is not specified to be a domain error in C, but it is defined as a domain error in POSIX.

POSIX also specifies that in case of underflow, arg is returned unmodified, and if that is not supported, an implementation-defined value no greater than DBL_MIN, FLT_MIN, and LDBL_MIN is returned.

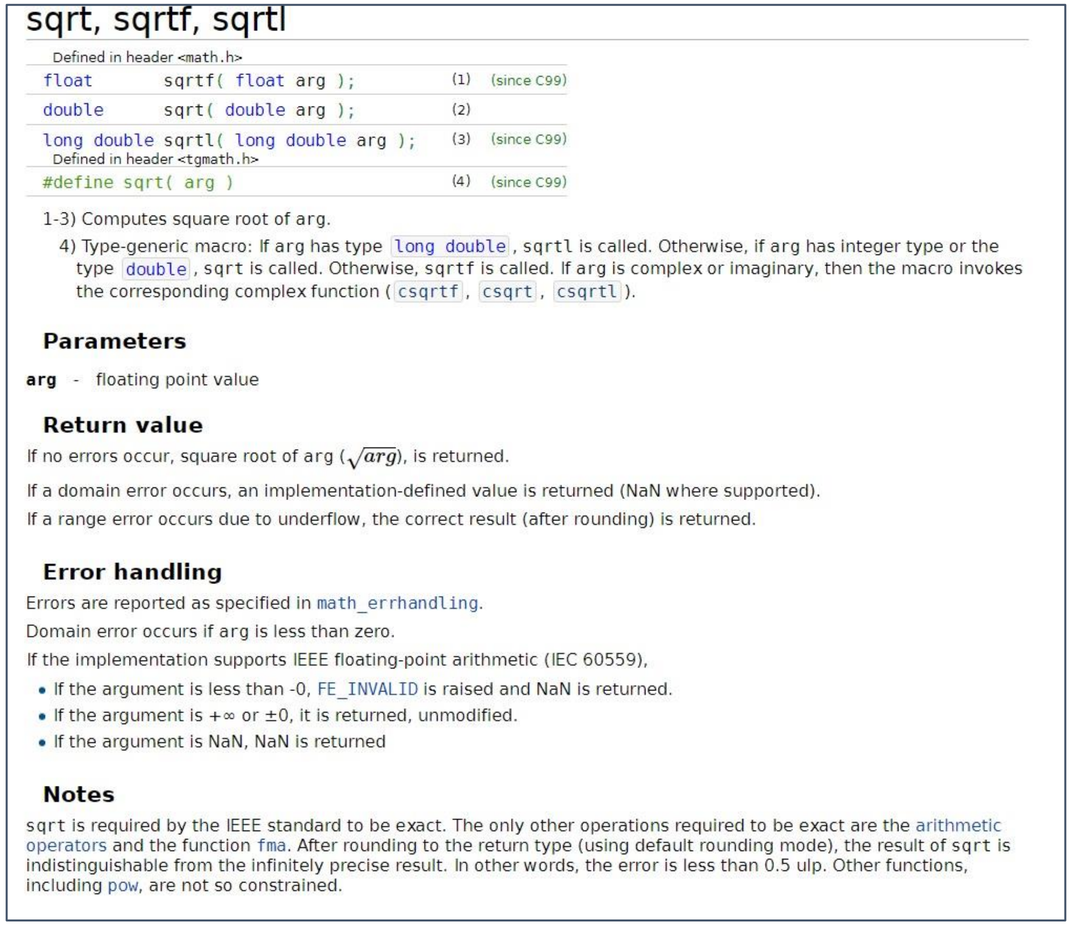
WHAT IS `SQRT`?

Part II

C reference says…

(http://en.cppreference.com/w/c/numeric/math/sqrt.)

- 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.



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 to 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.

C reference says…

(http://en.cppreference.com/w/c/numeric/math/sqrt.)

- Notes
  - `sqrt` is required by the IEEE standard to 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.

----

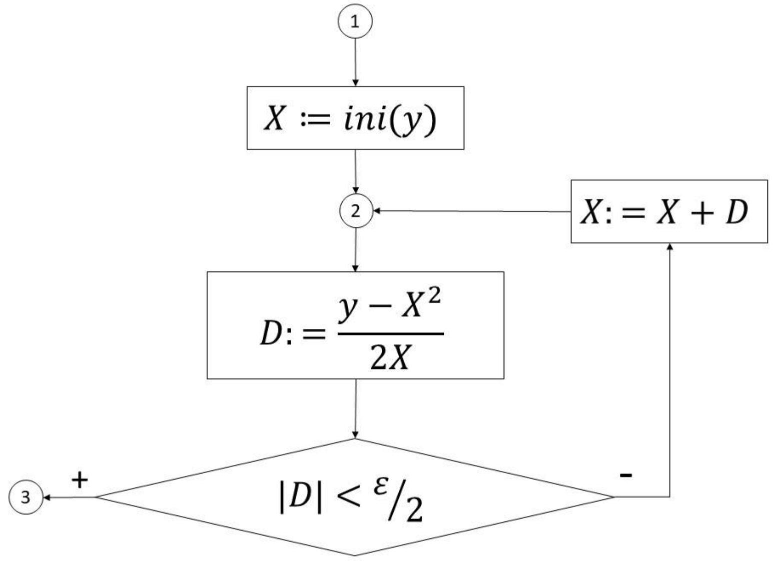
*) ulp stays for *Unit in the Last Place*

Alternatives for the standard `sqrt`

- It makes sense to introduce another function with two arguments $SQR(y, \varepsilon)$ where y stays for the argument and ε stays for accuracy (precision), that can be mathematically specified by the following clauses:
  - if $y \geq 0$ and $\varepsilon > 0$ then $SQR(y, \varepsilon)$ must return a (real) value $x \geq 0$ that differs from $\sqrt{y}$ not more than ε, i.e., *truncation error is* $|x - \sqrt{y}| \leq \varepsilon$.

How to compute SQR (in ideal real arithmetic)

- Input: Any non-negative real value $y \geq 0$ and a positive accuracy $\varepsilon > 0$.
- Algorithm implementing the Newton method (see right).
- Output: A real value x_n approximating $\sqrt{y}$ with accuracy ε.



SQR algorithm: specification, annotations and verification (in ideal real arithmetic)

- Specification:
  - precondition: $y > 1$(!) and $\varepsilon > 0$;
  - postcondition: $|x - \sqrt{y}| \le \varepsilon$;
  - Hoare triple: $[y > 1 \,\&\, \varepsilon > 0]SQR[|x - \sqrt{y}| \le \varepsilon$.
- Annotations (inductive assertions):
  1. precondition;
  2. invariant: precondition and $\sqrt{y} < x \le y$;
  3. postcondition.
- Verification: Floyd-Hoare proof method (for *partial* and *total* correctness).

А.П. Ершов – С.С. Лаврову (март 1983 г.)

Поздравляю с днем рождения, милый друг!
Шестьдесят уже как выпало на круг.
Завтра то ли, как вчера, себя вести,
То ли новую программу завести.

Чтобы грамотно программу составлять,
Надо пред- и постусловия задать,
Надо точный счетчик времени иметь
И циклический инвариант привлечь.

Предусловие — твой праздничный венец,
Постусловие — у всех один конец,
Время катится без помощи чужой,
Для инварианта — будь самим собой!

Computer fix-point arithmetic

Part III

Fix-point arithmetic axiomatic (with Gaussian rounding)

- Below we present version axiomatization (modulo ideal arithmetic of real and integer numbers) of a computer (platform-independent) fix-point arithmetic data type.
- Please remark that we explicitly admit that there may be several different fix-point data types simultaneously.

A fix-point data-type $\boldsymbol{D}$: values

- The set of values $Val_{\boldsymbol{D}}$ is a finite set of rational numbers Q such that it contains
  - the least $Inf_{\boldsymbol{D}} < 0$ and the largest $Sup_{\boldsymbol{D}} > 0$ elements,
  - all rational numbers in $[Inf_{\boldsymbol{D}}, Sup_{\boldsymbol{D}}]$ with a step $\delta_{\boldsymbol{D}}$
  - all integers $Int_{\boldsymbol{D}}$ in the range $[Inf_{\boldsymbol{D}}, Sup_{\boldsymbol{D}}]$.

A fix-point data-type $\boldsymbol{D}$: operations and relations

- Admissible operations include machine addition $\oplus$, subtraction $\ominus$, multiplication $\otimes$, division $\oslash$, integer rounding up $\lceil\quad\rceil$ and down $\lfloor\quad\rfloor$.
- Admissible binary relations include all standard equalities and inequalities (within $[Inf_{\boldsymbol{D}}, Sup_{\boldsymbol{D}}]$) denoted in the standard way ($=, \neq, <, >, \leq, \geq$).

Machine addition and subtraction.

- If the exact result of the standard mathematical addition (subtraction) of two fix-point values falls within the interval $[Inf_{\boldsymbol{D}}, Sup_{\boldsymbol{D}}]$,then machine addition (subtraction respectively) of these arguments equals to the result of the mathematical operation (and notation $+$ and $-$ is used in this case).

Machine multiplication and division

- Operations return values that are nearest in $Val_{\boldsymbol{D}}$ to the exact result of the corresponding standard mathematical operation: for any $x, y \in Val_{\boldsymbol{D}}$
  - If $x \times y \in Val_{\boldsymbol{D}}$ then $x \otimes y = x \times y$.
  - If $^{x}/_{y} \in Val_{\boldsymbol{D}}$ then $x \oslash y = {}^{x}/_{y}$.
  - If $x \times y \in [Inf_{\boldsymbol{D}}, Sup_{\boldsymbol{D}}]$ then $|x \times y - x \otimes y| \le {}^{\delta_{\boldsymbol{D}}}/_{2}$.
  - If$^{x}/_{y} \in [Inf_{\boldsymbol{D}}, Sup_{\boldsymbol{D}}]$ then $|{}^{x}/_{y} - x \oslash y| \le {}^{\delta_{\boldsymbol{D}}}/_{2}$.

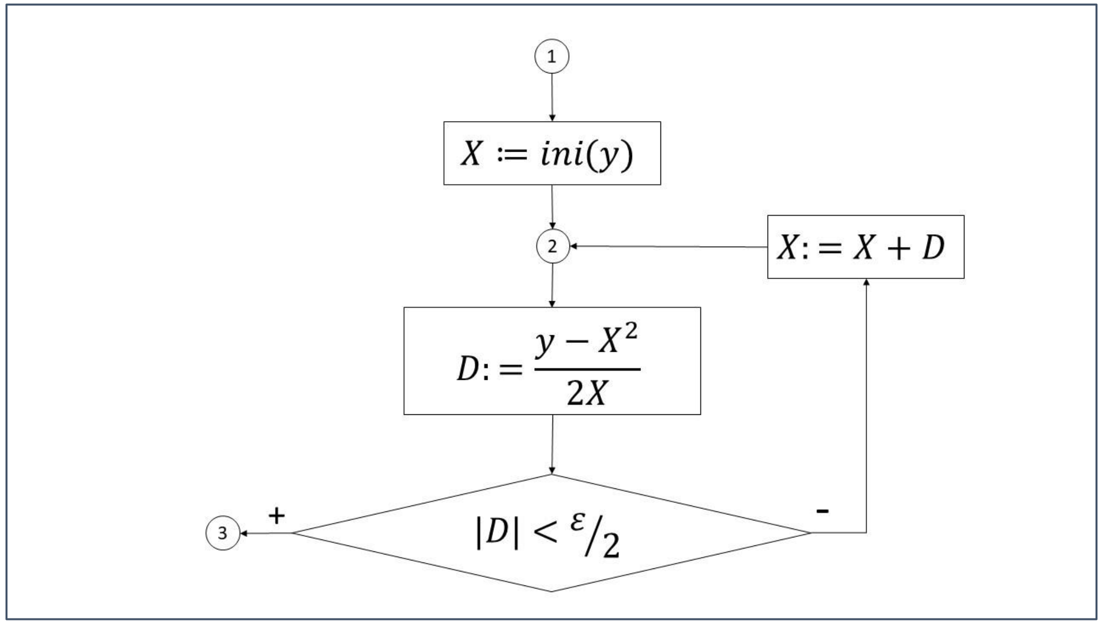
Verification of SQR in the fix-point arithetic

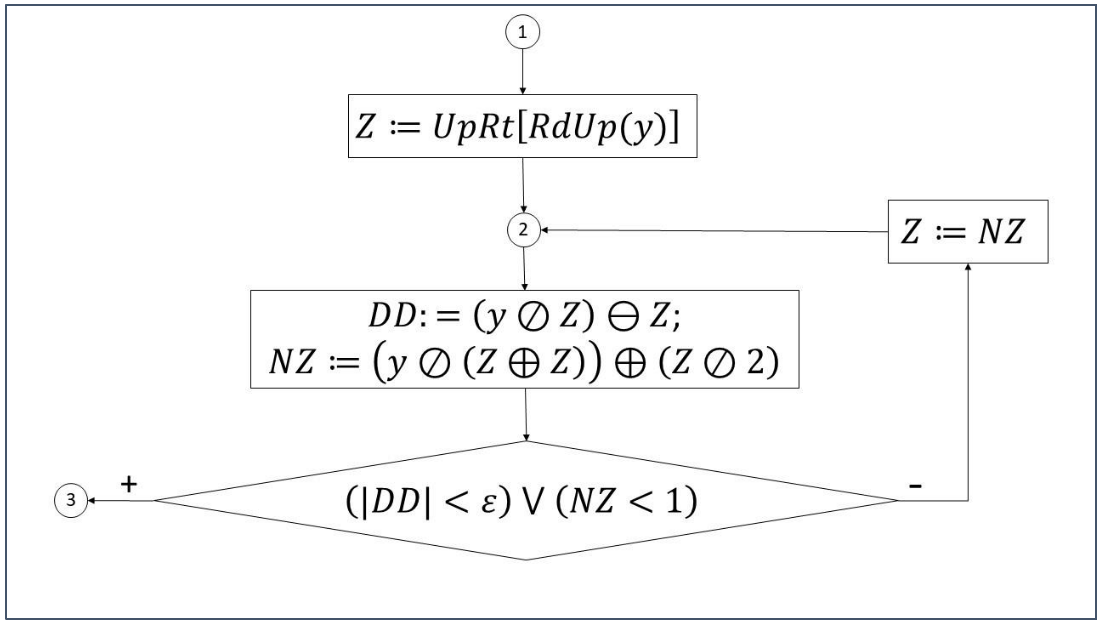
Part IV

Platform-independent approach

- In our approach we
  - start with easy-to-verify Hoare total correctness assertions for logical specification of imperative algorithms that implements the computer functions in ideal real arithmetic,
  - then move to pen-and-paper (manual) specification and verification of computer algorithm (with computer arithmetic) using the ideal case as proof outlines and guidance,
  - and finish with computer-aided verification of the computer functions in computer fix-point arithmetic.

Ideal vs. computer algorithm




1
$Z \coloneqq UpRt[RdUp(y)]$
2
$Z \coloneqq NZ$
$DD := (y \oslash Z) \ominus Z;$
$NZ \coloneqq \left(y \oslash (Z \oplus Z)\right) \oplus (Z \oslash 2)$
$(|DD| < \varepsilon) \vee (NZ < 1)$
+
-
3

Specification (manually verified)

$$
\left[\begin{array}{ll}
(a) & 1 < y \in Val_{\mathbb{D}} \ \& \\
(b) & \delta_{\mathbb{D}} \leq \frac{1}{12} \ \& \\
(c) & 5\frac{1}{6}\delta_{\mathbb{D}} < \varepsilon \in Val_{\mathbb{D}} \ \& \\
(d) & \sqrt{y} \leq UpRt\big[\lceil y \rceil\big] \leq \sqrt{y} + \frac{1}{2} \ \& \\
(e) & 2\left(\sqrt{y} + \frac{1}{2}\right) < \sup_{\mathbb{D}} - 2\delta_{\mathbb{D}}
\end{array}\right]
$$

$$LANIFAv2$$

$$\big[|\sqrt{y} - NZ| \leq \left(\varepsilon + 2\delta_{\mathbb{D}}\right)\big]$$

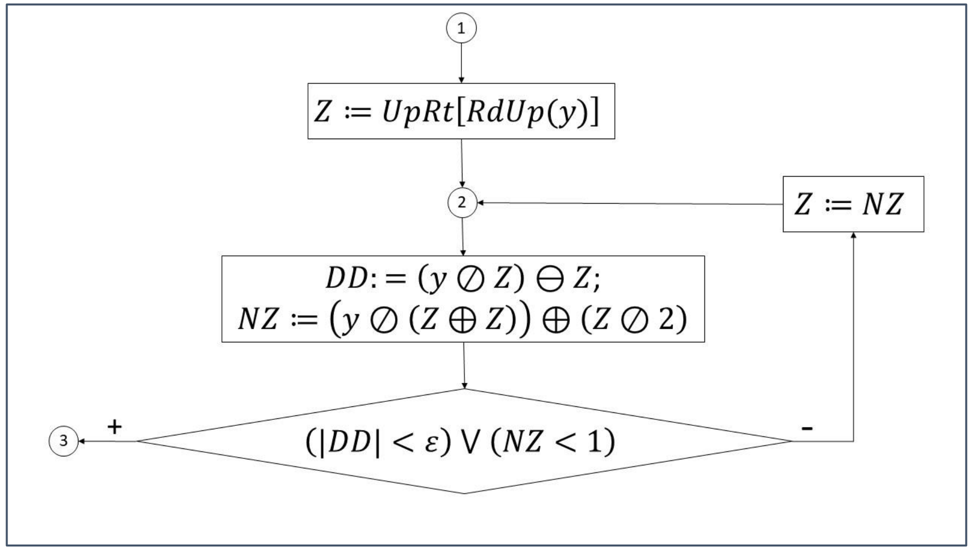
ACL2: A Computational Logic for Applicative Common Lisp

- ACL2 is an automated proof assistant based on applicative dialect of Common Lisp.
- Modules in ACL2 are called books. Each book comprises function definitions and theorems. Proving process of all theorems specified in a book is called certification.
- Any theory from a certified book may be included (imported) for further use to other books.

Towards computer-aided verification

- As now we have verified with ACL2 existence of the look-up table…

$$
\begin{array}{ll}
(a) & 1 < y \in Val_{\mathbb{D}} \ \& \\
(b) & \delta_{\mathbb{D}} \leq \frac{1}{12} \ \& \\
(c) & 5\frac{1}{6}\delta_{\mathbb{D}} < \varepsilon \in Val_{\mathbb{D}} \ \& \\
(d) & \sqrt{y} \leq UpRt\big[\lceil y \rceil\big] \leq \sqrt{y} + \frac{1}{2} \ \& \\
(e) & 2\left(\sqrt{y} + \frac{1}{2}\right) < \sup_{\mathbb{D}} - 2\delta_{\mathbb{D}}
\end{array}
$$


1
$Z := UpRt[RdUp(y)]$
2
$Z := NZ$
$DD := (y \oslash Z) \ominus Z;$
$NZ := \big(y \oslash (Z \oplus Z)\big) \oplus (Z \oslash 2)$
$(|DD| < \varepsilon) \vee (NZ < 1)$
+
-
3

Adding a (virtual) Platform

Part V

Better precision in programming languages

- Currently, tools to increase the precision of fix- and floating-point calculations are available in many industrial platforms (C / C ++, Java, Python), but their usability leaves much to be desired.
- For example, to work with numbers of unlimited bit depth in Java (BigInteger, BigDecimal), the programmer must explicitly call the methods of the corresponding class for each arithmetic operation. – With this approach, the computational formula cannot be written in natural notation, but must be decomposed into the corresponding elementary actions, which causes inconvenience and can serve as a source of errors.

A virtual platform for better precision

- This situation is not accidental: in the industrial languages, data of a non-standard type are represented by objects, and it takes effort to *link* them with the standard types.
- In contrast we deign and prototype a simple programming language to carry out calculations with arbitrary high specified precision (ulp).
- There are no problems with writing algebraic formulas in the standard way, since the language has built-in numeric types – fixed-point numbers and rational numbers, floating-point numbers with a definable mantissa size.

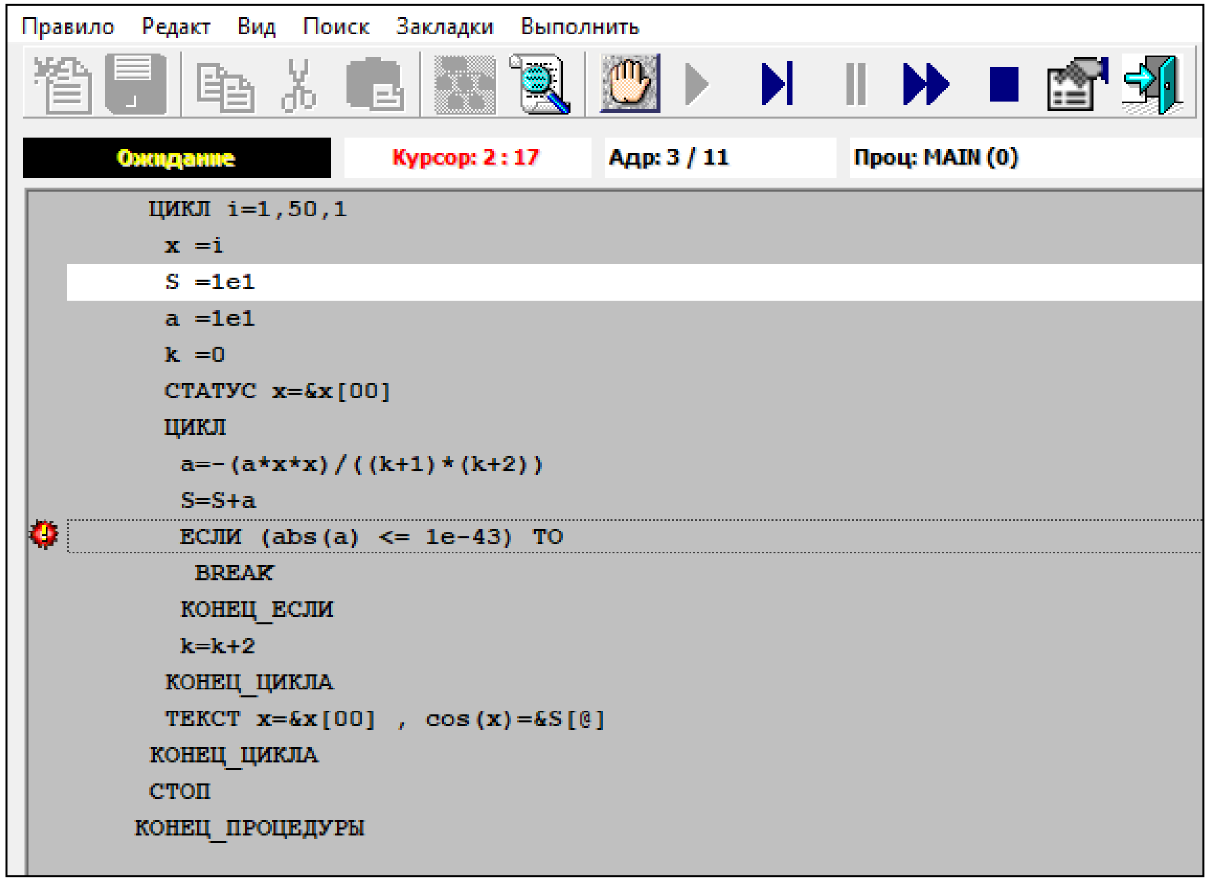
Example

```
_REM=Косинус с 45-разрядной мантиссой
_MSIZE=45
_OUT=cos-45flo.txt
 [BEGIN]
ПРОЦЕДУРА MAIN()
   ЦИКЛ i=1,50,1
       x =i
       S =1e1
       a =1e1
       k =0
       СТАТУС x=&x[00]
```

```
ЦИКЛ
          a=-(a*x*x)/((k+1)*(k+2))
          S=S+a
          ЕСЛИ (abs(a) <= 1e-43) ТО
             BREAK
          КОНЕЦ_ЕСЛИ
          k=k+2
       КОНЕЦ_ЦИКЛА
       ТЕКСТ x=&x[00] , cos(x)=&S[@]
   КОНЕЦ_ЦИКЛА
   СТОП
  КОНЕЦ_ПРОЦЕДУРЫ
```

Example – cont.

- x=47, cos(x)=-992335469150928718279752449751460894306084247e0
- x=48, cos(x)=-640144339469199731312941408847427594399083185e0
- x=49, cos(x)=300592543743637083687025835872943611527344800e0
- x=50, cos(x)=964966028492113274068957350623116893380516422e0



```
    ЦИКЛ i=1,50,1
     x =i
     S =1e1
     a =1e1
     k =0
     СТАТУС x=&x[00]
     ЦИКЛ
      a=-(a*x*x)/((k+1)*(k+2))
      S=S+a
      ЕСЛИ (abs(a) <= 1e-43) ТО
       BREAK
      КОНЕЦ_ЕСЛИ
      k=k+2
     КОНЕЦ_ЦИКЛА
     ТЕКСТ x=&x[00] , cos(x)=&S[@]
    КОНЕЦ_ЦИКЛА
    СТОП
   КОНЕЦ_ПРОЦЕДУРЫ
```

Why do we need a platform?

- We are going to use the proposed software to test “verified” standard functions, as well as a tool for laboratory assignments in mathematics (and not only in computational mathematics).