Journal of Universal Computer Science, vol. 4, no. 2 (1998), 147-163
submitted: 25/9/97, accepted: 1/11/97, appeared: 28/2/98

Constructive Error Analysis

Walter Krämer
(Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung
(IWRMM), Universität Karlsruhe, 76128 Karlsruhe, Germany
E-mail: walter.kraemer@math.uni-karlsruhe.de)

Abstract: Rigorous a priori error bounds for floating-point computations are derived. We will show that using interval tools in combination with function and operator overloading such bounds can be computed on a computer automatically in a very convenient way. The bounds are of worst case type. They hold uniformly for the specified domain of input values. That means, whenever the floating point computation is repeated later on with any set of point input values from that domain the difference of the exact result and the computed result is guaranteed to be smaller than the a priori error bound.
Our techniques can be used to get reliable a priori error bounds for already existing program code. Here, loops, recursion, and iterations are allowed. To demonstrate the power of the methods several examples are given.
Key Words: Scientific computing, Validated numerics, Rigorous error bounds, Floating point computations

1 Introduction

Let a mathematical expression $e(x;p)$ be given. x denotes the vector of independent variables and p the set of parameters. The result of the computation of $e(x;p)$ at the vectors of machine numbers $\tilde{x}, \tilde{p}$ already afflicted with some errors using floating point arithmetic is denoted by $\tilde{e}(\tilde{x};\tilde{p})$. Then we are interested in the computation of an a priori error bound $\Delta(e)$ for the absolut error $|e(x;p)-\tilde{e}(\tilde{x};\tilde{p})|$ which holds for all $x \in [x], |x-\tilde{x}| \leq \Delta(x)$ and all $p \in [p], |p-\tilde{p}| \leq \Delta(p)$, simultaneously. $[x]$ and $[p]$ are interval vectors which define the ranges of x and p, respectively. $|\,.\,|$ is to be applied componentwise, and $\Delta(x), \Delta(p)$ are nonnegative vectors.

Several different approaches (error bound arithmetics for absolute errors, for relative errors, error factor arithmetics for absolute/relative error factors) are discussed. In any case the main idea is to use the recursive definition of an expression (such an expression may be given by a program part) only using basic operations like addition, subtraction, multiplication, division, sine, cosine, logarithm, and other elementary functions. For the basic operations we will discuss the computation of error bounds for the results of the corresponding machine operations applied to arguments that are already afflicted by errors. The computation of such bounds can be done using interval arithmetic and, in case of the elementary functions, using automatic differentiation with interval arguments. In general we only rely on the so called $(1+\varepsilon)$ property of floating point operations.

> A t-digit radix R floating-point arithmetic possesses the $(1+\varepsilon)$ property if, for all floating-point numbers a and b and $\circ \in \{+,-,\cdot,/\}$, there exists ε such that the machine result $a \boxed{\circ}\, b = (a \circ b)(1+\varepsilon)$ and $|\varepsilon| < R^{1-t}$, provided no underflow or overflow occurs and $b \neq 0$ if $\circ = /$.

For example, the IEEE 754 floating-point arithmetics [8] all possess this property. This is true for any one of the four rounding modes.

We will also demonstrate, that intermediate results in the underflow range (here the $(1+\varepsilon)$ property is not valid) can be treated by our tools in a reliable way using some simple additional considerations. So our results are valid without the warning "provided no underflow occurs".

Let us demonstrate the main idea by a very simple example:

$$e(x) := \mathrm{ln1p}\left(x + \frac{x}{\sqrt{1+\left(\frac{1}{x}\right)^2} + \frac{1}{x}}\right)$$

Here $\mathrm{ln1p}(x)$ denotes the real valued function $\mathrm{ln1p}(x) := \ln(1+x)$, $x > -1$. The given expression may be represented by the following computation tree:

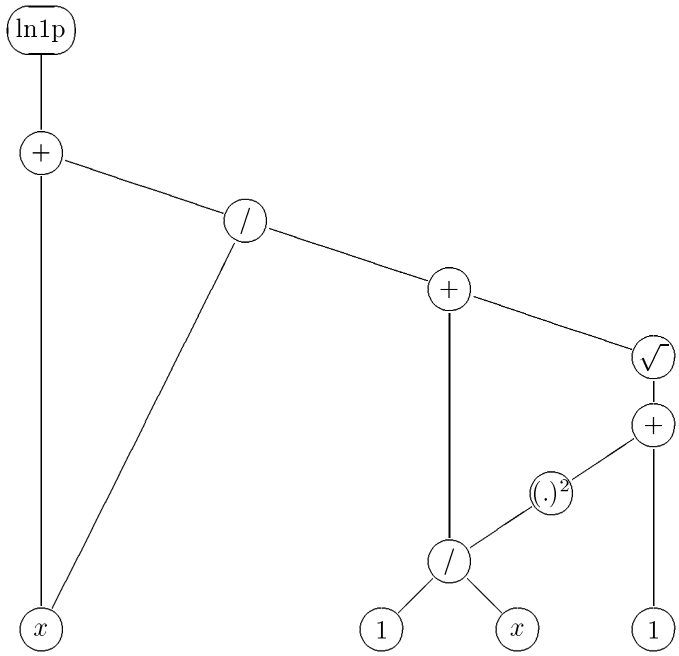

Figure 1: Ordinary Computational Tree

To evaluate this expression on a machine an ordered list of basic floating-point operations is applied to intermediate results starting with the floating-point equivalents $\tilde{x}, \tilde{p}$ of the numerically known data x, p appearing in the leaves of the tree. (Here p denotes the exactly representable constant 1. In general constants of a computation must be approximated by numbers representable in the floating-point screen. They are regarded as inputs.)

To be able to construct a reliable worst-case error bound for $\tilde{e}(\tilde{x}, \tilde{p})$ we introduce a new data type. Each variable of this type has two components, the first one being an interval and the second one a real value greater than or equal to zero. The first component encloses the exact value of the variable and the second one represents an error bound for the corresponding floating point quantity.

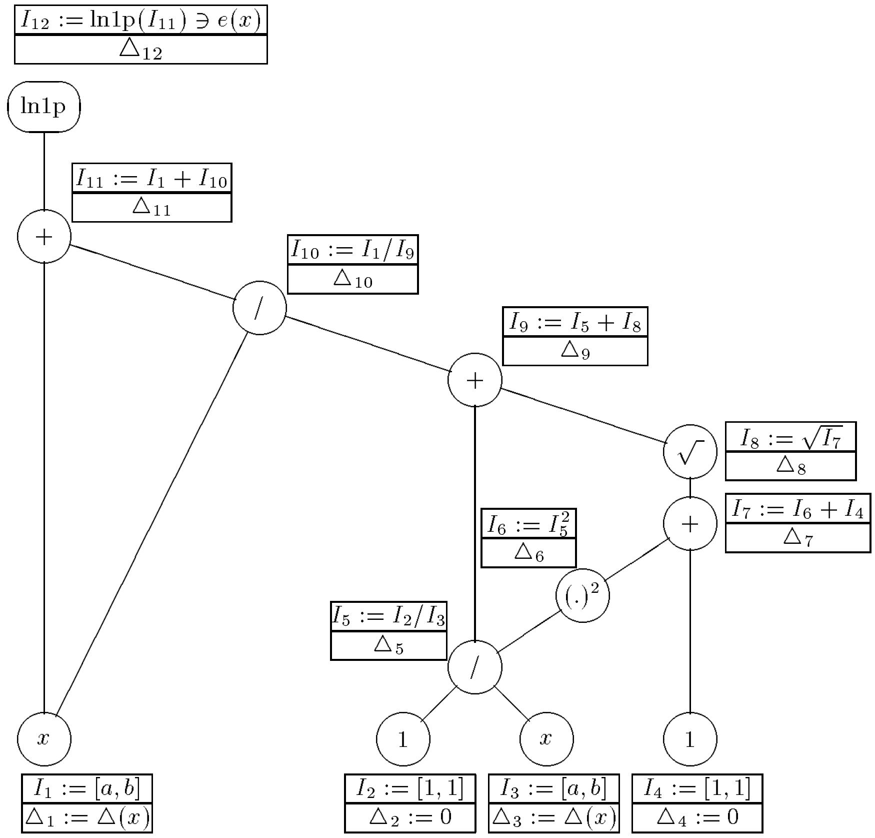

Figure 2: Computational Tree With Function and Operator Overloading

Using the results for the error bounds of the basic operations makes it possible to overload the ordinary operators und functions for variables of the new data type. Such a new operation computes an enclosure of the actual intermediate result as well as an error bound valid for this intermediate result. In the preceding diagram (see Figure 2) variables of the new data type are represented by rectangles with two entries. The first entry indicates the interval enclosure and the second the corresponding error bound.

All quantities in the bottom line are known (domain $[x] := [a, b]$ of x, absolute error bound $\Delta(x)$ with $|x - \tilde{x}| \leq \Delta(x)$, error bounds for the parameters, which for this example are all 0). Computing the expression step by step from the bottom line to the root results finally in the interval enclosure I_{12} of the exact range $\{e(x) : x \in [x]\}$ and the error bound Δ_{12} with $|e(x) - \tilde{e}(\tilde{x})| \leq \Delta_{12}$. It should be emphasized, that the enclosures of the intermediate results are only computed to find at the end of the process the final error bound Δ_{12}. Indeed, we are not interested in $I_k,\ k = 0, 1, 2\ldots$ itself. For the computation of the individual bounds Δ_j the enclosures I_k are used as auxiliary quantities.

At the end of the process we know the worst case error bound Δ_{12}. It tells us that for any subsequent floating point computation of $\tilde{e}(\tilde{x})$ with floating point argument $\tilde{x}$ with $|x - \tilde{x}| \leq \Delta(x)$ for some $x \in [x]$ the difference $|e(x) - \tilde{e}(\tilde{x})|$ will be less than or equal to this bound. In practical applications narrow bounds can be achieved even for quite large domains $[x]$ (see the numerical examples in Section 5, and [6]).

Note: the computation time for the error bounds is (nearly) irrelevant, because the computation is performed only <u>once</u>.

2 Error Bounds for Basic Operations Only Using the $(1 + \varepsilon)$ Property

In this section we give no proofs. They can be found in [6, 7, 11, 12].

We denote by $S = S(R, l)$ the screen of floating point numbers with l radix R mantissa digits. $\circ \in \{+, -, *, /\}$ denotes a basic operation and $\boxed{\circ}$ the corresponding floating point operation. $MinReal$ means the smallest normalized positive floating point number, $MaxReal$ the largest one and $\varepsilon = R^{1-l}$ (Wilkinson's epsilon). By A and B (intervals) we denote the ranges of the operands a and b, respectively.

To be able to find rigorous error bounds for the basic operations applied to already disturbed arguments we assume the so called $(1 + \varepsilon)$ property (see also the introduction). I. e., for the arithmetic operations applied to floating point numbers a and b we assume

Assumption I)

$$|a \circ b| \ \in \ [\, MinReal,\ MaxReal\,]$$

$$\Longrightarrow \left| \frac{a \circ b \ - \ a \boxed{\circ}\, b}{a \circ b} \right| \leq \varepsilon, \text{ i. e. } a \boxed{\circ}\ b = (1 + \text{eps})(a \circ b)\,, \quad |\text{eps}| \leq \varepsilon$$

For results lying in the range of denormalized numbers or in the underflow range we additionally assume

Assumption II) (Underflow)

$$|a \circ b| \leq MinReal \Longrightarrow |a \circ b \ - \ a \boxed{\circ}\, b| \leq MinReal$$

Again, IEEE 754 floating point arithmetics all possess this property

Let the floating point numbers $\tilde{a}, \tilde{b}$ be approximations to the exact values a, b, respectively. We now give error bounds for the basic arithmetical operations:

Addition: If we define the quantity $\Delta(\text{add})$ by

$$\Delta(\text{add}) := MinReal + \varepsilon \cdot |A + B| + (1 + \varepsilon)\Big(\Delta(a) + \Delta(b)\Big)$$

we can show that

$$(*) \quad a \in A,\ |a - \tilde{a}| \le \Delta(a), \quad b \in B,\ |b - \tilde{b}| \le \Delta(b)$$
$$\Longrightarrow |a + b \ - \ \tilde{a} \boxplus \tilde{b}| \le \Delta(\mathrm{add})\,.$$

We want to emphasize that the bound $\Delta(\mathrm{add})$ holds uniformly for all pairs $(a, \tilde{a})$ and $(b, \tilde{b})$ which fulfill $(*)$.
Multiplication: The error bound for the multiplication is given by

$$\Delta(\mathrm{mul}) := MinReal + |A||B|\varepsilon +$$
$$(1+\varepsilon)\,(|A|\Delta(b) + |B|\Delta(a) + \Delta(a)\Delta(b))$$

Division: Here we denote by $\langle B\rangle$ the real number $\langle B\rangle := \min\{\,|b| : b \in B\,\}$ (mignitude). We additionally assume that

$$\Delta(b) < \ 0.5 \ \cdot \langle B\rangle\,.$$

For division we define the quantity $\Delta(\mathrm{div})$ by

$$\Delta(\mathrm{div}) := MinReal + \frac{1}{\langle B\rangle - \Delta(b)} \cdot \Big(\ \Delta(a) +$$
$$(|A| + \Delta(a)) \cdot (\varepsilon + \frac{\Delta(b)}{\langle B\rangle} + 2(\frac{\Delta(b)}{\langle B\rangle})^2) \ \Big).$$

Again, this bound holds simultaneously for all pairs $(a, \tilde{a})$ and $(b, \tilde{b})$ with $a \in A,\ |a - \tilde{a}| \le \Delta(a), \quad b \in B,\ |b - \tilde{b}| \le \Delta(b)$, that is we have for arbitrary such pairs

$$|a/b \ - \ \tilde{a} \boxed{/} \tilde{b}| \le \Delta(\mathrm{div})\,.$$

So far we are able to find error bounds for arithmetical expressions composed of basic arithmetical operations. But we also want to allow elementary function computations with uncertain data. Let f denote an elementary function (e. g. sin, cos, arctan, log, ...) and $\tilde{f}$ its machine equivalent. We assume that $f : \mathbb{R} \supseteq D_f \to \mathbb{R}$ is continuously differentiable in $A + [-\Delta(a),\ \Delta(a)]$. We further assume that for a valid floating point argument x the machine version $\tilde{f}$ produces a result with a relative error which is bounded in the following way:

$$|f(x) - \tilde{f}(x)| \le |f(x)|\,\varepsilon(f)\ , \ \ x \in S\,.$$

That means, we assume that we already know a relative error bound for $\tilde{f}$ which holds uniformly for all exactly representable arguments. Because the concrete implementation $\tilde{f}$ can be interpreted as an arithmetical expression built from the four basic operations, such a bound $\varepsilon(f)$ can be derived applying our results from above concerning the error propagation of the basic arithmetical operations.

Error bound for function evaluation: If the quantity $\Delta(f)$ is computed by

$$\Delta(f) := \varepsilon \cdot |f(A)| + (1+\varepsilon(f)) \cdot \Delta(a) \cdot |f'([A-\Delta(a),\, A+\Delta(a)])|$$

then it holds for any pair $(a, \tilde{a})$ with $a \in A,\ \tilde{a} \in S,$ and $|a-\tilde{a}| \leq \Delta(a)$

$$|f(a) - \tilde{f}(\tilde{a})| \leq \Delta(f)\,.$$

Again this bound holds uniformly for all valid pairs $(a, \tilde{a})$. The derivative f' can be computed using the process of automatic differentiation. If this process is done in interval arithmetic an enclosure for the range of values of the derivative over an interval can be computed easily.

We want to emphasize that all error bounds given above are uniform error bounds. They are valid whenever the exact (point) arguments lie in their prescribed ranges and the corresponding floating point approximations are within their prescribed bounds. The formulae for the error bounds can be computed using directed rounded operations and/or interval operations. All quantities appearing in the formulae are easily accessible on a computer.

3 Further Improvements in Case of a Faithful Arithmetic

In some important situations the error bounds from the previous section can be improved. For example, if we try to find an error bound for the process of argument reduction in an elementary function subroutine. A sharp error bound is essential to finally get a good over all error bound for the complete algorithm.

We first want to identify when computed differences are exact. Such differences are important because they do not, by themselves, introduce additional errors.

To be able to get stronger results on error bounds we assume from now on that the floating point arithmetic is faithful. This means:

> If the exact result is representable, the result of the corresponding flp operation is equal to that number. If not, the floating point result is one of the two flp numbers nearest the exact result.

Let us notice that almost any commercially significant arithmetic is faithful. Of course, any IEEE-754 compliant arithmetic is faithful.

In the following we want to give two theorems concerning exact differences.

Sterbenz (see [17]): If a and b are two floating point numbers such that

$$\frac{1}{2} \leq \frac{a}{b} \leq 2\,,$$

then in a faithful arithmetic $a \boxminus b = a - b$ exactly.

Ferguson (see [4]): Let x and y be machine numbers for which the significand $s(x)$ has $z(x) \geq 0$ trailing zeroes, $s(y)$ has $z(y)$ trailing zeros, and the exponent

$$e(x-y) \leq \min\{e(x)+z(x),\, e(y)+z(y)\}\,.$$

Then the computed value $x \boxminus y$ is exact if floating point subtraction is faithful.
More such theorems can be found in [4, 17, 15].

Exact multiplication (exponent manipulation): A faithful floating point multiplication is error free if one of the factors is a power of the radix (and no underflow or overflow occurs).

Exact multiplication (short significands): Let the sum of trailing zeros of the significands of two t-digit floating point numbers x and y be greater than or equal to t. Then $x \boxdot y = x \cdot y$, exactly (if no underflow or overflow occurs) and if $\boxdot$ is faithful.

In the following section we will discuss software implementations for an error bound arithmetic which take advantage of the preceding theorems.

4 Software Tools

The following code segment shows a PASCAL-XSC [9] implementation to find automatically an error bound for the floating point division operator applied to floating point data afflicted by errors.

For an interval X we define MinAbs(X):= $\min\{|x| : x \in X\} = \langle X \rangle$ and MaxAbs(X):= $\max\{|x| : x \in X\} = |X|$.

```
global function DeltaDiv(
           alpha,  beta  : interval;
           DeltaA, DeltaB: real      ): real;
var x: real;
begin
  ...  S P E C I A L  C A S E S ...
  else begin
    x:= DeltaB /> MinAbs(beta);
    if x >= 0.5 then  ... E R R O R ...
    x:= Eps +> x +> 2*>x*>x;
    x:= DeltaA +> x *> ( MaxAbs(alpha) +> DeltaA );
    DeltaDiv:=
       MinReal +> x /> ( MinAbs(beta)  -< DeltaB );
  end;
end;
```

With

$$A \subseteq \mathtt{A}, \Delta(a) \leq \mathtt{DeltaA}, B \subseteq \mathtt{B}, \Delta(b) \leq \mathtt{DeltaB}$$

the function call

```
bound:= DeltaDiv(A, B, DeltaA, DeltaB);
```

returns the machine number `bound` with

$$a \in A,\ |a - \tilde{a}| \leq \Delta(a),\ b \in B,\ |b - \tilde{b}| \leq \Delta(b)$$

$$\Longrightarrow \left| a/b - \tilde{a} \boxed{/} \tilde{b} \right| \leq \Delta(\mathrm{div}) \leq \texttt{bound}$$

The usage of such a function (or similar functions for other basic operations) is cumbersome. The usual mathematical notation of expressions has to be transformed to corresponding sequences of function calls. Existing code has to be modified drastically.

To avoid this we introduce a new data type called `BoundType` and overload the basic operators for this new data type:

```
GLOBAL TYPE BoundType = GLOBAL RECORD
   Value : Interval;
   AbsErr: real;
END;
```

The first component of a variable of this type is an enclosure of the range of values of the corresponding real variable. The second component is a reliable absolute error bound for the corresponding floating point quantity.

In PASCAL-XSC the division operator can now be overloaded for operands of the new data type

```
GLOBAL OPERATOR / (x, y: BoundType) res: BoundType;
BEGIN
   res.Value :=  x.Value / y.Value;
   res.AbsErr:=
   DeltaDiv(x.Value, y.Value, x.AbsErr, y.AbsErr);
END;
```

Similarly, the subtraction operator can be overloaded by

```
global operator - (x, y: BoundType) res: BoundType;
var zx, zy: integer;
begin
  res.Value := x.Value - y.Value; {ordinary interval subtraction}

  if ((x.AbsErr=0) and (y.AbsErr=0)) then begin
    { trailing zeros of significand(x): }
    if x.Value.inf = x.Value.sup then
      zx:= tz_test(x.Value.inf)
    else
      zx:= 0;

    ...

    if expo(MaxAbs(res.Value)) <=
            min(  expo(MinAbs(x.Value)) + zx,
                  expo(MinAbs(y.Value)) + zy   )
    then
       res.AbsErr:= 0
    else
       res.AbsErr:= DeltaAdd(x.Value, -y.Value, 0.0, 0.0);
  end else begin
```

```
  res.AbsErr:= DeltaAdd(x.Value, -y.Value, x.AbsErr, y.AbsErr);
end;
```

The previous code segment shows the realization of a tool for the automatic computation of error bounds for faithful subtractions (see the theorem of Sterbenz cited in Section 3).

The overloaded operations now can be used to compute error bounds for expressions and program parts. The second diagram in the introduction shows how the new operators manipulate variables of the new data type `BoundType`. The two components of such variables are shown in this diagram as rectangles subdivided in two parts. The upper subbox indicates the range of values of the actual intermediate result, the lower one indicates its worst case absolute error bound. Arriving at the top of the diagram we know that an over all error bound for the complete expression is given by the numerical value of the quantity Δ_{12}.

5 Applications, Numerical Examples

In this section the tools from the previous section are applied to get reliable error bounds for different simple problems. We will discuss two point problems as well as a problem with a very large domain for its input data. In [6, 7, 11] more examples are given. There you can find an application of the tools described above to socalled table-driven algorithms for elementary functions ([19, 16, 14]).

5.1 Example 1, Summation in Different Directions

Let us first consider a simple summation. We want to add the first terms of the Taylor expansion for the exponential $\exp(x) = \sum \frac{x^k}{k!}$. We compare a summation from left to right $(\ldots(((1+x)+x^2/2)+x^3/3!)+\ldots)+x^n/n!$ with the mathematical equivalent summation from right to left $1+(x+(x^2/2+(x^3/3!+(\ldots+(x^n/n!)\ldots))))$. We are interested in worst case error bounds for the two different methods.

We show program parts for the different summation schemes:

```
function SumLeftToRightErr(x: BoundType; n: integer): real;
var k: integer;
    s, sk: BoundType;
begin
  s := exact(1);    { 1 + x + x/2 + x^2/6 + x^3/24 + ... }
  sk:= exact(1);
  for k:= 1 to n do begin
     sk:= sk*x/exact(k);
     s := s + sk;
  end;
  SumLeftToRightErr:= s.AbsErr/Eps52; {Error in multiples of eps}
end;

function SumRightToLeftErr(x: BoundType; n: integer): real;
var k: integer;
    s, sk: BoundType;
```

```
begin
  s := exact(0);
  for k:= n downto 1  do begin
     s:= (s+exact(1))*(x/exact(k));
  end;
  s:= s+exact(1);
  SumRightToLeftErr:= s.AbsErr/Eps52; {Error in multiples of eps}
end;
```

At the point $x = 1/8$ we get for different numbers of summands the following numerical results for the two methods (`eps` denotes the machine precision, `exact(.)` means, that its argument is an integer or a floating point number which is error free):

```
Number of summands=  6
   LeftToRight:  4.541 Eps
   RightToLeft:  1.292 Eps
Number of summands= 11
   LeftToRight:  10.206 Eps
   RightToLeft:   1.292 Eps
Number of summands= 16
   LeftToRight:  15.872 Eps
   RightToLeft:   1.292 Eps
Number of summands= 21
   LeftToRight:  21.538 Eps
   RightToLeft:   1.292 Eps
Number of summands= 26
   LeftToRight:  27.204 Eps
   RightToLeft:   1.292 Eps
```

The summation from left to right adds the summands in decreasing order, whereas the second method adds the smaller summands first. Now it is well known from numerical analysis that the second method should give better error bounds for the over all rounding error. Indeed, using our error bound arithmetic we find numerical values which show the expected behaviour.

5.2 Example 2, Multi-Precision Computations

Here we want to perform an error estimation for a simple iterative process. The iteration is intended to give approximates to π to high accuracy. We start the iteration with

$$a_0 := \sqrt{2}, \;\; b_0 := 0, \;\; p_0 := 2 + \sqrt{2}.$$

The iterates are computed using the following formulae

$$\left.\begin{aligned} a_{n+1} &:= \frac{1}{2}\left(\sqrt{a_n} + \frac{1}{\sqrt{a_n}}\right) \\ b_{n+1} &:= \sqrt{a_n}\,\frac{1+b_n}{a_n+b_n} \\ p_{n+1} &:= p_n\, b_{n+1}\,\frac{1+a_{n+1}}{1+b_{n+1}} \end{aligned}\right\} n = 0, 1, \ldots$$

According to [2] a bound for the relative error for the n-th approximate p_n to π is given by

$$\left|\frac{\pi - p_n}{\pi}\right| \leq \frac{1}{2} 10^{1-2^n} .$$

This especially means that the number of correct digits is doubled in each iteration step (exact calculations in the field of real numbers are assumed). The error propagation coming from (multi-precision) floating point calculations is not covered so far. We have to answer questions like

> How many guard digits have to be used to get a result which is accurate to a specified number of digits? If we want to compute π to say one million decimal places is it sufficient to carry out the iterations with say one million and five digits?

To see how powerful and elegant our proposed method is you should compare its application with a hand calculation of an error bound for the first say 30 iteration steps.

The automatic error estimation is done by the following PASCAL-XSC program (notice, that the program code is only a slight modification of the corresponding code to perform the multi-precision computation of π). Here we use the so called error factor arithmetic (see Section 6) with its new data type `AbsErrType`.

```
program PiAbsErr;
{----------------------------------------------------------------------}
{ Error factors for a quadratically convergent iteration to compute pi }
{----------------------------------------------------------------------}
use i_ari;      { Ordinary interval arithmetic                         }
use abs_ari;    { Absolute error factor arithmetic                     }

var
  An, Bn, Pn      : AbsErrType;                       { Old iterates }
  Anp1, Bnp1, Pnp1: AbsErrType;                       { New iterates }
  zp5, two        : AbsErrType;               { Constants  0.5 and 2 }
  n, nMax         : integer;                  { Iteration counter    }
  Eps             : string;                   { String '*Epsilon'    }
begin
  Eps:= '*Epsilon ';
  two:= 2;                                   { Exactly representable }
  zp5:= 0.5;                                 { Exactly representable }
```

```
  Anp1:= sqrt(two);   { = a0 }                     { Initialization }
  Bnp1:= 0;           { = b0 }
  Pnp1:= 2 + anp1;    { = p0 }
  writeln('   n        Absolute Error Bounds for  An, Bn and Pn');
  writeln;
  nMax:= 32;
  for n:= 1 to nMax do begin                            { Iteration }
     An:= Anp1;
     Bn:= Bnp1;
     Pn:= Pnp1;
     Anp1:= zp5*(sqrt(An) + sqrt(1/An)); { Computation of new iterates }
     Bnp1:= sqrt(An)*(1 +Bn)/(An + Bn);
     Pnp1:= Pn*Bnp1*(1 + Anp1)/(1 + Bnp1);
     writeln( n: 4, '  ',  Anp1.AbsErr:9:1:1, Eps,  Bnp1.AbsErr:9:1:1,
              Eps, Pnp1.AbsErr:9:1:1, Eps);
   end;
 end.
{----------------------------------------------------------------------}
```

The output of the program is as follows:

```
  n         Absolute Error Bounds for  An, Bn and Pn

  1        3.8*Epsilon       3.8*Epsilon      46.7*Epsilon
  2        5.2*Epsilon      13.0*Epsilon     131.5*Epsilon
  3        5.9*Epsilon      23.1*Epsilon     265.1*Epsilon
 ...           ...              ...                ...
 31        6.6*Epsilon     343.7*Epsilon   25858.4*Epsilon
 32        6.6*Epsilon     355.2*Epsilon   27558.1*Epsilon
```

Interpretation of the Result:

Assuming a multi-precision arithmetic with $2^{32}+3$ decimal places, i. e.

$$\varepsilon := \frac{1}{2} 10^{1-(2^{32}+3)}$$

we get

$$\left|\frac{\pi - \tilde{p}_{32}}{\pi}\right| \leq \left|\frac{\pi - p_{32}}{\pi}\right| + \left|\frac{p_{32} - \tilde{p}_{32}}{\pi}\right|$$

$$\leq \frac{1}{2} \cdot 10^{1-2^{32}} + \frac{27559 \cdot \varepsilon}{\pi} \ \leq \ \frac{1}{2} \cdot 10 \cdot 10^{1-2^{32}} \ .$$

The error bound calculations show that

$$\pi \in [\tilde{p}_{32} - \frac{1}{2} 10^{2-2^{32}}, \ \tilde{p}_{32} + \frac{1}{2} 10^{2-2^{32}}] \, .$$

So **three** guard digits are enough to be sure that at most **one** additional digit (with respect to the inherent approximation errors) is lost! Using a multi-precision arithmetic with $2^{32}+3$ decimal digits we will get after 32 iterations an approximation to π which is guaranteed to be correct to $2^{32}-1 =$ 4 294 967 295 decimal places. Notice, that the error bounds are computed using an ordinary floating point (interval) arithmetic. No multi-precision calculations are performed to find the error factors.

5.3 Example 3, Expression for the Inverse Hyperbolic Sine

The inverse hyperbolic sine function can be computed using the formula

$$\operatorname{arsinh}(x) := \operatorname{ln1p}\left(x + \frac{x}{\sqrt{1+\left(\frac{1}{x}\right)^2} + \frac{1}{x}} \right) \tag{1}$$

Here, $\operatorname{ln1p}(x) := \ln(1+x)$. The formula has already been used in the introduction of this paper. Numerically, this formula is only appropriate in a restricted domain. To cover the full domain (all IEEE-754 double numbers) the following representation with $z := |x|$ is used

$$\operatorname{arsinh}(x) \approx \operatorname{sign}(x) \cdot \begin{cases} z, \ z \in [0, 2.5E-8] \\ \operatorname{ln1p}(z + \frac{z}{\sqrt{1+(\frac{1}{z})^2} + \frac{1}{z}}), \ z \in [2.5E-8, 1.25] \\ \ln(z + \sqrt{z^2+1}), \ z \in [1.25, 10^{150}] \\ \ln 2 + \ln(z), \ z \in [10^{150}, \mathrm{MaxReal}] \end{cases}$$

The break points shown here are adequate for the IEEE double format. The following program part is the core routine to compute a *relative* error bound for the expression in the second line of the representation given above.

```
use  i_ari,                            { Interval arithmetic    }
     abs_ari;                          { Error bound arithmetic }

function MaxError(y: interval; n: integer): real;
var res, h, x: BoundType;
    eMax: real;
begin
   eMax:= 0;
   for i:=1 to n do begin
    x:= subinterval(i,y);       { Subdivision into n subregions }
    x.AbsErr:= 0;         { Exactly rep. positive flp arguments }
    h:= 1/x;
    res := ln1p( x  + x / ( sqrt( 1+ h*h ) + h ));
    eMax:= max( eMax, res.AbsErr /> MinAbs(res.Value) );  { (*) }
  end;
  MaxError:= eMax;
end;
```

We want to compute a simultaneous error bound which holds for all floating point arguments (that means the complete real axis). For this purpose we have to subdivide the domain into smaller subregions. This is done by a driver program. Using the relation

$$\operatorname{arsinh}(-x) = -\operatorname{arsinh}(x)$$

reduces the domain of interest to the positive real axes (negation is error free). In the driver program a suitable subdivision of the positive real axis is done. The complete domain is covered by about 10000 individual subregions. The width of the subregions is not uniform. Some individual bounds for the subregions are shown in the output of the program given below.

Our error bound arithmetic intrinsically computes absolute error bounds. But here we are interested in a relative bound. To avoid a division by 0 in the program line indicated by `(*)`, we restrict the argument range for the automatic error bound calculation to the interval $[2.5E-008, 1.79E+308]$ which, of course, is still a very large interval. Our calculation gives the following relative bounds over the different subregions (Formula (1) is only used in the interval $[2.5E-8, 1.25]$, outside this interval other formulas are more appropriate; the over all error bound given below is valid for the corresponding mixture of formulas):

```
           Subregion              Relative error bound
[ 2.49999E-008, 1.00000E-007]        4.7428330E-016
              ...                         ...
[ 1.00000E-003, 1.00001E-001]        5.2015858E-016
[ 1.00000E-001, 1.00000E+000]        5.6460092E-016
[ 1.00000E+000, 1.25000E+000]        5.6160721E-016
              ...                         ...
[ 9.99999E+304, 9.99999E+305]        5.1753451E-016
[ 9.99998E+305, 9.99999E+306]        5.1752958E-016
[ 9.99998E+306, 1.79769E+308]        5.1795241E-016
```

The over all relative error bound for the domain $[2.5E-8, 1.79E+308]$ is as follows:

```
Computed relative error bound   5.6460092E-016
```

An error bound for the domain $[0,\, 2.5E-8]$ is very easily derived by hand. For values $0 < |x| \leq 2.5E-8$ the approximation $\operatorname{arsinh}(x) \approx x$ is used (i. e. no rounding error occurs). We find

$$\left|\frac{\operatorname{arsinh}(x) - x}{\operatorname{arsinh}(x)}\right| \leq \frac{\frac{1}{2\cdot 3}x^2}{1 - \frac{1}{2\cdot 3}x^2} \leq 1 \cdot \varepsilon \,.$$

In summary we now know

$$|\operatorname{arsinh}(x) - \widetilde{\operatorname{arsinh}}(x)| \;\leq\; |\operatorname{arsinh}(x)| \cdot 2.6 \cdot \varepsilon \,,\; x \in S \,. \qquad (2)$$

Here $\varepsilon := 2^{-52}$ is the approriate machine precision covering any rounding mode of an IEEE double arithmetic. If we assume round to nearest operations, the given error bound can be improved significantly.

We want to emphasize that the computed error bound holds for the concrete arsinh floating point implementation. Manipulating the implementation in any way (e. g. modifying the numerical values of the break points or modifying the sequence of operations) in general results in a different over all error bound (as we have already seen in Example 1 above).

For the concrete implementation we now know a worst case error bound. Whenever we compute for a floating point number x the value $\widetilde{\operatorname{arsinh}}(x)$ (we assume that this value is not in the underflow range) we can be sure that the difference to the exact function value arsinh(x) is bounded as described by formula (2). Using this information and the monotonicity of the inverse hyperbolic sine makes it easy to implement an arsinh-version that accepts a floating point interval X as argument and gives back an enclosure of the correct range of values over X.

6 Different Kinds of Error Bound Arithmetics

In this paper we only have discussed in some detail the so called error bound arithmetic for absolute errors: Using the known quantities $A, B, \Delta(a), \Delta(b)$ we have found e. g. an error bound $\Delta(\text{add})$ in such a way that

$$a \in A,\ |a - \tilde{a}| \leq \Delta(a), \quad b \in B,\ |b - \tilde{b}| \leq \Delta(b)$$

$$\Longrightarrow |a + b\ -\ \tilde{a} \boxplus \tilde{b}| \leq \Delta(\text{add})\ .$$

A modified approach is the so called error *factor* arithmetic. The absolute error bounds are written as products. Only the so called error factor is treated as a numerical value. To get the numerical value of the corresponding bound this factor has to be multiplied by the numerical value $\varepsilon := 0.5 \cdot R^{1-l}$ (precision of the computer arithmetic for which the floating point algorithm is applied). The quantity R denotes the base (radix) of the floating point screen and l the number of mantissa digits.

The error factor arithmetic is well suited to find error bounds for multi-precision computations (such an arithmetic has been used to estimate the rounding errors in Example 2 above). The quantities A, B as well as the error factors $k(a)$ for the left and $k(b)$ for the right operand are given. The factor $k(\text{add})$ is to be computed in such a way that (see [11])

$$a \in A,\ |a - \tilde{a}| \leq k(a) \cdot \varepsilon, \quad b \in B,\ |b - \tilde{b}| \leq k(b) \cdot \varepsilon$$

$$\Longrightarrow |a + b\ -\ \tilde{a} \boxplus \tilde{b}| \leq k(\text{add}) \cdot \varepsilon\ .$$

Another possibility is an error bound arithmetic for *relative* errors. A, B as well as the relative error bounds $\varepsilon(a), \varepsilon(b)$ of the operands are known and we want to find a relative error bound $\varepsilon(\text{add})$ with

$$a \in A,\ \tilde{a} = a \cdot (1 + \varepsilon_a),\ |\varepsilon_a| \leq \varepsilon(a),$$

$$b \in B,\ \tilde{b} = b \cdot (1 + \varepsilon_b),\ \ |\varepsilon_b| \leq \varepsilon(b)$$

$$\Longrightarrow \tilde{a} \boxplus \tilde{b} = (a + b) \cdot (1 + \varepsilon_{add}),\ |\varepsilon_{add}| \leq \varepsilon(add)\ .$$

For all kinds of such error arithmetics we can derive explicit formulas for the quantities we are looking for. So it is possible to implement reliable software tools using an ordinary interval arithmetic as well as directed rounded floating point operations. Again, operator and function overloading can be used to get very comfortable and user-friendly tools. Existing programs can be analysed in a very simple way. In essence, only the type of the variables has to be adapted and some initialization has to be performed.

7 Conclusion

We have shown that our approach can be used to get worst case error bounds for data error and/or rounding error propagation in floating point computations almost automatically. Due to operator and function overloading our software tools can be applied to existing program code in a very comfortable way. Loops, recursion, and iterations are allowed. Conditional statements must be handled with some care: for point data the sequence of operations is unique whereas the specification of input data by domains may destroy this property. In some special cases an appropriate domain subdivision may resolve this problem.

We have used the technique of the automatic generation of error bounds for floating point algorithms extensively to get reliable error estimates in the field of mathematical function implementations. Meanwhile we have implemented a new and fast elementary function library with known reliable worst case error bounds. The functions accept real and interval arguments. The routines are based on the double precision floatig-point format and the corresponding basic operations as defined by the IEEE standard 754 [8]. The portable ANSI-C code of our function routines is available via ftp.

The work described in this paper is still in progress. It should be possible to incorporate ideas coming for example from [3, 13, 17, 4, 15]. See also Section 3. Up to now our software tools are written in PASCAL-XSC but we are also preparing a C++ class library. The source code of our tools will be made available electronically.

8 Acknowledgement

I want to thank Werner Hofschuster for fruitful discussions about the topics of this paper.

References

1. Bauer, F. L.: *Computational Graphs and Rounding Error.* SIAM J. Numer. Anal. **11**, 87 –96, 1974.
2. Borwein, J.M., Borwein, P.B.: *The arithmetic-geometric mean and fast computation of elementary functions*, SIAM Review 26, 1984, 351-366.
3. Dekker, F. D. : *Floating-Point Technique for Extending the Available Precision.* ACM Trans. Math. Soft. 5, 204–217, 1979.
4. Ferguson, W. E.: *Exact Computation of a Sum or Difference with Applications to Argument Reduction.* Proceeding to the 12th Symposium on Computer Arithmetic, IEEE Computer Society Press, 216–221, 1995.
5. Higham, N. J.: *Accuracy and Stability of Numerical Algorithms.* SIAM, 1996.
6. Hofschuster, W., Krämer, W.: *Ein rechnergestützter Fehlerkalkül mit Anwendungen auf ein genaues Tabellenverfahren*, Preprint 96/5 des IWRMM, Universität Karlsruhe, 1996.
7. Hofschuster, W., Krämer, W.: *A Computer Oriented Approach to Get Sharp Reliable Error Bounds*, Reliable Computing, Issue 3, Volume 3, pp 239-248, 1997.
8. American National Standards Institute / Institute of Electrical and Electronics Engineers: *A Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std. 754-1985, New York, 1985 (reprinted in SIGPLAN **22**, 2, pp 9–25, 1987).

9. Klatte, R., Kulisch, U., Neaga, M., Ratz, D., Ullrich, Ch.: *PASCAL–XSC — Language Reference with Examples*. Springer-Verlag, Berlin/Heidelberg/New York, 1992.
10. Krämer, W.: *Fehlerschranken für häufig auftretende Approximationsausdrücke*. ZAMM **69**, pp T44–T47, 1989.
11. Krämer, W.: *Eine Fehlerfaktorarithmetik für zuverlässige a priori Fehlerabschätzungen.* Institut für Angewandte Mathematik, Universität Karlsruhe, Forschungsschwerpunkt Computerarithmetik, Intervallrechnung und Numerische Algorithmen mit Ergebnisverifikation, 5/1997.
12. Krämer, W.: *A Priori Worst Case Error Bounds for Floating Point Computations.* Proceedings of the 13th Symposium on Computer Arithmetic, edited by Tomas Lang, Jean-Michel Muller, Naofumi Takagi, IEEE Computer Society, 64–71, 1997.
13. Linnainmaa, S.: *Software for Doubled-Precision Floating-Point Computations.* ACM Trans. Math. Soft. 7, 272–283, 1981.
14. Muller, J.-M.: *Elementary Functions: Algorithms and Implementation.* Birkhäuser, Boston, 1997.
15. Priest, D. M.: *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations.* Thesis, 1992.
16. Priest, D. M.: *Fast Table-Driven Algorithms for Interval Elementary Functions.* Proceedings of the 13th Symposium on Computer Arithmetic, edited by Tomas Lang, Jean-Michel Muller, Naofumi Takagi, IEEE Computer Society, 168–174, 1997.
17. Sterbenz, P. H.: *Floating-Point Computation.* Prentice-Hall, Series in Automatic Computation, Englewood Cliffs, N. J. 1974.
18. Stoer J., Bulirsch, R.: *Introduction to Numerical Analysis.* Springer–Verlag, New York, Heidelberg, Berlin, 1983.
19. Tang, P. T. P.: *Table-Driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic*. ACM Trans. on Math. Software, Vol. **15**, No. 2, pp 144–157, 1989.
20. Wilkinson, J: *Modern Error Analysis.* SIAM Review 13, 548–568, 1971.