Fixed precision patterns for the formal verification of mathematical constant approximations

Yves Bertot
Inria
Yves.Bertot@inria.fr

Abstract

We describe two approaches for the computation of mathematical constant approximations inside interactive theorem provers. These two approaches share the same basis of fixed point computation and differ only in the way the proofs of correctness of the approximations are described. The first approach performs interval computations, while the second approach relies on bounding errors, for example with the help of derivatives. As an illustration, we show how to describe good approximations of the logarithm function and we compute π to a precision of a million decimals inside the proof system, with a guarantee that all digits up to the millionth decimal are correct. All these experiments are performed with the Coq system, but most of the steps should apply to any interactive theorem provers.

1. Introduction

Modern higher-order logic proof systems are usually well-suited to describe real numbers and to reason about computations based on this type of number. However, when it comes to comparing numbers, it is often useful to have approximations that make it easy to show how the comparison should be solved, with only exceptions occurring when the comparison is tight, where pure mathematical reasoning remains the surest solution.

A principled approach to this question is to first study a mathematical algorithm that approximates the intended real computation, and then to study how this mathematical algorithm is refined taking into account the fact that most elementary operations are not performed exactly, but only approximately. Thus, the same mathematical object can be represented by three values: *(i)* the exact real number x (often the limit of an infinite sequence of other real numbers x_n), *(ii)* a real number approximating x expressed using only elementary operations (often one of the elements of the infinite sequence, say x_p for some fixed p), *(iii)* an approximated version the real value, as described in a number system that only covers a denumerable subset of the real numbers (an approximation of x, computed with rounded elementary operations).

For instance, we may consider the constant π. Among the three values mentioned in the previous paragraph, π itself plays the role *(i)*. This constant may be computed using Machin's formula $\pi = 4 \times (4 \arctan(\frac{1}{5}) - \arctan(\frac{1}{239}))$. Each of the arctangent function calls (the function noted atan) can then be approximated as the limit described by the following definition:

$$\arctan(x) = \lim_{n \to \infty} \sum_{i=0}^{n} \frac{(-1)^i x^{2i+1}}{2i+1}$$

Putting everything together, we know that

$$4 \times (4 \times (\frac{1}{5} - \frac{1}{3 \times 5^3}) - \frac{1}{239})$$

is an under approximation of π. This value plays the role *(ii)*. When computing a decimal representation of this number, we can choose to use decimal representations for the various terms in the sum, like

$$\frac{1}{3 \times 5^3}.$$

But for this term, no finite decimal representation is exact, so that the final value that we compute when using 5 decimal places, 3.14059 is again an under approximation. This value plays the role *(iii)*.

Thus, there are two approximations being performed in the process, each of them introduces an *approximation error* and we want to prove properties of these errors. In this paper, we will focus mostly on the second part: once we know a mathematical process, using only elementary operations (addition, subtraction, division, multiplication, and square root), how can we represent this process using concrete computations in an interactive theorem prover, so that we obtain an easy to use approximation of this constant. All our techniques will rely on integer computations and assume that we have a library to perform these computations in a formally verified way. In a first section, we will show the approach we advocate to perform high precision computation, which practically boils down to fixed point computation.

We then will describe two approaches. In the first approach, we will reproduce *interval* analysis at the level of each elementary operation. Each value will be known through a formally verified upper bound and a formally verified lower bound, and we will use bounds on inputs to compute bounds on outputs. In the second approach, we will only assume that each input is known within a given error and we will study how this error is modified by successive operations. In particular, we will see that we can prove tighter bounds with the second approach than with the first one, because it makes it possible to avoid the weakness of interval analysis known as the *dependency* or *decorrelation* problem.

2. Related work

Similar questions have been studied in a variety of proof systems over the last decade. A notable publication concerning the PVS system, by Daumas, Lester, and Muñoz [8] completely describes

interval arithmetics for most of the mathematical functions. These researchers prefered to use rational numbers to describe the bound of intervals. We also experimented with this approach, one advantage is that multiplication and division are exact. On the other hand, numerators and denominators of fractions tend to get very big. To keep them small, greatest common divisor computations need to be added, and this is costly.

Melquiond also worked on interval approaches, with experiments in the PVS system [9] (with Daumas and Muñoz) and the Coq system [14]. An advantage of this work is that it systematically uses interval splitting to reduce the effect of decorrelation. For the sake of completeness, we can illustrate here how to compute the natural logarithm of 10 using Melquiond's tool. We first compute iterated square roots on 10, to obtain a number smaller than 2, we then use a series expansion of the logarithm function. To conclude we need a proof of correctness for that series expansion (but this is not the point of this paper).

```
Lemma ssss10 :
 2130196773249681 6844/18446744073709551616
             <= sqrt(sqrt(sqrt(sqrt 10))) <=
 2130196773249681 6845/18446744073709551616.
interval with (i_prec 128).
Qed.

Lemma upper_ln_10_div_16 :
  forall x,
  2130196773249681 6844/18446744073709551616
      <=  x <=
  2130196773249681 6845/18446744073709551616 ->
  2654699870154338133/18446744073709551616 <=
  sum_f_R0 (fun n => (-1)^n * (x - 1) ^
              (n + 1)/INR(n + 1)) 10
<=
  2654699870154338135/18446744073709551616.
Proof.
intros; simpl; interval with (i_prec 128).
Qed.
```

This example only provides an upper bound for $\ln \sqrt[16]{10}$ as a rational number whose denominator is a power of 2, a lower bound can be computed in a similar way. This example shows that the first method described in this paper (where we also use interval computations) is practically already covered by Melquiond's `interval` tactic. The most obvious restriction in the `interval` tactic is that it only works with powers of 2 as denominators of bounds. The work we describe in this paper around the logarithm function is slightly more general, since we obtain approximations with an arbitrary denominator.

Real numbers in Coq have also been studied extensively in the `CCoRN` library, with a strong constructive philosophy. In the constructive approach, every real number actually describes an algorithm that can give more and more precision as required. Obviously, this approach makes it possible to compute mathematical constants at an arbitrary precision. However, constructive real numbers are cumbersome to use from a pure classical point of view, because the equality of real numbers is not decidable (for instance, this precludes the definition of piece-wise continuous functions). A nice extension of this work is provided by Kaliszyk and O'Connor [12], who show that efficient implementations of real numbers in their constructive setting can also be used to reason about real numbers in the classical setting, which is the context of our work. Unfortunately, we have not been able to re-use their work, because it was not maintained continuously.

The `Flyspeck` project, concerned with the optimal placement of spheres in three-dimensional space is another setting where computations of real numbers play a significant role. A big part of this project is the verification of a large collection of inequalities for multi-variate functions, including most polynomial expressions, arctangent, square roots and constants like π. Obviously, a tool to verifiy inequalities would also be able to provide bounds for mathematical constants. Solovyev and Hales [17] developed a specific tool for this need, and we expect it to be usable for moderate precision requirements, but it is explicitly stated in Solovyev's thesis [16] that the tool is not tuned for extremely high precision. This development is done in HOL-Light [11], which takes a more minimalist approach with respect to arithmetic computations than Coq. In Coq the big integers are computed using binary integer arithmetic directly for 31 bit integers.

This paper also describes (if only partially) the formally proved computation of π to a precision of the order of 10^{10^6}, inside the Coq theorem prover. Most modern proof systems contain a definition of π and a tool to compute it to a high precision, but rarely exceeding 10^{100}. Usually, the technique relies on a *Machin*-like formula[1] The algorithm we use in this paper is based on arithmetic-geometric means and was probably described for the first time in [6]. There is another, better known, algorithm also based on arithmetic-geometric means, published simultaneously and independently by Brent [7] and Salamin [15]. That other algorithm is the one used in the high-precision, high-assurance, library MPFR [18]. Our execution time is still several orders of magnitude lower than in MPFR.

3. Fixed point computation

If we want to perform fixed point computation in decimal notation, it means that all numbers have a decimal point and a fixed number of digits to the right of the decimal point. For instance, we may decide to perform all computations with 5-digit after the decimal point. Numbers will have the following form: 1.00000 (this represents 1), 1.23456, (a number between 1 and 2), -4.22567 (a number between -5 and -4).

When it comes to adding two numbers with five digits, we can forget about the decimal point. We just start by adding the rightmost digits, then the next to rightmost digits with the carry, and so on. At the end, we only need to check that we again obtain numbers where the decimal point is 5 places from the right. In fact, we might as well just use integers, but remember that each integer actually represents a fractional number. This number is just obtained by dividing by 10^k, where k is simply the number of digits on the right of the decimal point. In our running example, k is 5.

Thus, we use plain addition between integers, but each number n actually is used to represent another real number $[\![n]\!]$. This introduces a function $[\![\cdot]\!]$, actually defined by the following property:

$$[\![n]\!] = \frac{n}{10^k}.$$

In what follows, we will call *magnification ratio* the number 10^k, and we will forget about its shape and we will write μ in the mathematical formulas. Actually, it does not really matter that the number μ is a power of some base, and the number k will not occur anymore in our reasoning (but in our running example, we only need to remember that $\mu = 10^5$). We have the following definition for the brackets:

$$[\![n]\!] = \frac{n}{\mu}$$

and conversely

$$n = [\![n]\!] \times \mu.$$

[1] John Machin computed a hundred decimals of π using series expansions of the $\arctan$ function, in 1706.

Addition commutes nicely with brackets $[\![\cdot]\!]$. To add two real numbers represented by integers, we can simply add the two integers and then place the decimal point. This is expressed by the followint property.

$$[\![n]\!] + [\![m]\!] = [\![n+m]\!]$$

When remembering the meaning of the brackets $[\![\cdot]\!]$, this is just the following mathematical equality:

$$\frac{n}{\mu} + \frac{m}{\mu} = \frac{n+m}{\mu}.$$

For multiplication, $n \times m$ is too big to represent $[\![n]\!] \times [\![m]\!]$, by the order of magnitude of μ. This is illustrated by the following equality:

$$n \times m = [\![n]\!] \times \mu \times [\![m]\!] \times \mu.$$

If $[\![n]\!]$ and $[\![m]\!]$ are close to 1, then $[\![n]\!] \times [\![m]\!]$ should also be close to 1, and should thus be represented by an integer close to μ. However, $n \times m$ is close to μ^2. Moreover, $[\![n]\!] \times [\![m]\!]$ cannot usually be represented exactly.

For instance, with 5 digits after the decimal point, we can consider the two numbers 1.00001 and 1.00002 and multiply them together. The exact result is 1.0000300002 and can only be approximated if using numbers with 5 digits after the decimal point. One of the simplest solutions is to take the largest representable number that is smaller than the exact value. When considering positive values, this simply corresponds to truncating the extra digits.

This rounding down mechanism is simply implemented by using an integer division, where the divisor is μ. When the input is positive, we simply keep the quotient. When the input is negative we have to be careful and check the exact form of division provided for integers in the formal library. Sometimes, the division is designed to always return a non-negative remainder; sometimes the division is designed to return the same absolute value for the quotient, with a negative remainder if the dividend is negative. In the latter case and with a negative dividend, the quotient can be returned exactly only when the remainder is zero, in all other cases, the quotient minus one must be returned.

Of course, rounding towards zero is only one of the possible choices; we could alternatively choose to round towards minus infinity (always down, even for negative values) or towards plus infinity (always up). In the course of this paper, we actually have uses for each of these possibilities.

To summarize our treatment of multiplication, let's just introduce a new notation, $\cdot \overset{d}{\otimes} \cdot$ for multiplication with rounding towards minus infinity. Mathematically, this can be represented as follows:

$$n \overset{d}{\otimes} m = \left\lfloor \frac{n \times m}{\mu} \right\rfloor.$$

When it comes to division, the same kind of reasoning can be performed, if we just divide n by m and keep the quotient, we do not obtain a number that has the right magnitude to represent $\frac{[\![n]\!]}{[\![m]\!]}$. To understand this, it is simpler to write the mathematical formula:

$$\frac{n}{m} = \frac{[\![n]\!] \times \mu}{[\![m]\!] \times \mu}.$$

The two multiplications by μ cancel out in the fraction, while we were aiming at obtaining an integer close to the number

$$\frac{[\![n]\!]}{[\![m]\!]} \times \mu.$$

The solution is to multiply by μ, but this should be done before the division, to make sure that we are not multiplying the rounding error.

For instance, when we consider the division of 1.00000 by 3.00000, if we divide the first integer, 100000, by the second 300000, as an integer division, we obtain a quotient of 0, and multiplying by μ still remains at 0. On the other and, if we first multply by μ and then divide by 300000, we obtain 33333, the representation for 0.33333, the good truncated approximation of $\frac{1}{3}$.

To summarize our treatement of division, we can introduce the notation $\cdot \overset{d}{\oslash} \cdot$ for division with rounding towards minus infinity.

$$n \overset{d}{\oslash} m = \left\lfloor \frac{n \times \mu}{m} \right\rfloor.$$

Similar considerations apply for square roots. In particular, there exist algorithms for computing square roots of integers and some of these algorithmes have been implemented and formally verified in proof verification systems [2, 3]. It is then possible to implement rounded square root for fixed point computation. In this paper the notation will be $\overset{\lfloor\rfloor}{\sqrt{\cdot}}$, and defined as follows:

$$\overset{\lfloor\rfloor}{\sqrt{n}} = \left\lfloor \sqrt{n \times \mu} \right\rfloor.$$

All the rounded operations we have considered so far are rounded down, but we will also consider operations rounded up, noted $\cdot \overset{u}{\otimes} \cdot$, $\cdot \overset{u}{\oslash} \cdot$, $\overset{\lceil\rceil}{\sqrt{\cdot}}$. These operations are defined in a similar way.

We have already mentioned that addition can be implemented directly using the addition of integers, similarly, multiplication by an integer constant can also be implemented directly by integer multiplication: the integer multiplier should be used directly (not multiplied by the magnification ratio). Also, we should be careful that multiplying an upper bound of a given value by a negative integer will yield a lower bound.

4. Interval computations

We intend to use the various rounded operations (on integers) to represent elementary operations on real numbers, mostly to compute upper and lower bounds of approximations for polynomial expressions. These polynomial expressions are themselves truncated power series corresponding to the definition of mathematical functions.

4.1 Computation setting

We simply need to compute the polynomial expressions with a suitable replacement of the elementary operations by the corresponding integer operations, using rounded up or rounded down approximations as imposed by the sign of the polynomials coefficients.

A particularly interesting case of truncated power series is the case of alternating series, because we know that each truncation is either a lower bound or an upper bound of the expected value. For instance, for the logarithm function, standard mathematical reasoning makes it possible to obtain the following approximation (for $1 \leq x$):

$$\sum_{n=0}^{2k+1} (-1)^n \frac{(x-1)^{n+1}}{n+1} \leq \ln x \leq \sum_{n=0}^{2k} (-1)^n \frac{(x-1)^{n+1}}{n+1}.$$

So, to compute upper and lower bound of $\ln x$ for x representable in our fixed point setting, we should simply compute these polynomial expressions using our rounded elementary operations. For instance, we can compute a lower bound the third order expansion in the following way.

$$(x-1) - ((x-1) \overset{u}{\otimes} (x-1)) \overset{u}{\oslash} 2 + (x-1) \overset{d}{\otimes} (x-1) \overset{d}{\otimes} (x-1) \overset{d}{\oslash} 3.$$

Modern proof systems usually provide language constructs to make recursive definitions available and efficient ways to execute these recursive functions. We can use this ability to define lower and upper bounds of power functions and lower and upper bounds of expansions of power series. In the following, we will use the notations $\cdot \overset{l}{\uparrow} \cdot$ and $\cdot \overset{u}{\uparrow} \cdot$ to represent these lower and upper bound power functions.

When one wishes to compute bounds for $f(x)$, for x not representable in our fixed point setting, it is necessary to use the monotonicity of the function f in a neighborhood of x. This usually relies on extra information known about the mathematical function f. When f is known to be increasing (resp. decreasing) in a neighborhood of x, one should first find two representable numbers x_1 and x_2 in that neighborhood, such that $x_1 \leq x \leq x_2$ can be proved, and then one only needs to compute a lower bound (resp. upper bound) for $f(x_1)$ (resp. $f(x_2)$) and an upper bound (resp. lower bound) for $f(x_2)$ (resp. $f(x_1)$) to obtain lower and upper bounds of $f(x)$.

For instance to compute an upper bound of $\sin\sqrt{2}$, we only need to know that the $\sin$ function is increasing between 0 and $\frac{\pi}{2}$ and $\sqrt{2} < 1.42 < \frac{\pi}{2}$ and to compute an upper bound of $\sin(1.42)$.

4.2 Proving bounds

When it comes to proving properties, the main lesson is that we should organize the proof in two stages. In the first stage, we look at the algorithm from the point of view of real numbers and abstract properties of the rounded operations, in the second stage will look at operations on integers and we explain how these operations fulfill the abstract operations. For each of these stages, we define a function that describes exactly the algorithm being verified. We prove the abstract properties in the first stage, using real numbers, and then we show that the second function performs all the same computations but in a concrete setting.

4.3 First stage: with real numbers

We should state sufficient properties expected for our rounded functions and only reason using these properties. For our rounded operations, the properties we have chosen to use are the following ones:

$$0 \leq a \wedge 0 \leq b \Rightarrow 0 \leq a \overset{d}{\otimes} b \leq a \times b$$

$$0 \leq a \wedge 0 \leq b \Rightarrow a \times b \leq a \overset{u}{\otimes} b$$

$$0 \leq a \wedge 1 \leq b \Rightarrow a \overset{d}{\oslash} b \leq \frac{a}{b}$$

$$0 \leq a \wedge 1 \leq b \Rightarrow \frac{a}{b} \leq a \overset{u}{\oslash} b$$

These properties are more restrictive than the ones explained informally in section 3. The positivity constraints come from the fact that we decided to avoid questions about how integer division is defined for negative inputs; the constraint about the divisor being larger than 1 is probably an overkill, but it suits the fact that we are mostly interested in series with rational coefficients, so we will only divide by positive integers.

A first proof we can make (by induction on the power) is that the rounded power operations carry over the same properties as the multiplication: the rounded down power of a non-negative value is a lower bound of the power of that value (and is non-negative) and the rounded up power is an upper bound.

For the logarithm function, we describe a function that takes as input a number and a rank (stating how many terms of the series should be computed) and returns a triplet, where the first component is the lower bound of the truncated series at that input number, the second component is the upper bound, and the third component is boolean value that states what was the sign of the last term in the series. The recursive definition is as follows:

$$\begin{aligned}
\texttt{ln_i}(x, 0) &= (x, x, true) \\
\texttt{ln_i}(x, r+1) &= (a - (x \overset{u}{\uparrow} r+2) \overset{u}{\oslash} (r+2), \\
&\quad\; b - (x \overset{d}{\uparrow} r+2) \overset{d}{\oslash} (r+2), \mathit{false}) \\
&\quad\; \text{when} \quad \texttt{ln_i}(x, r) = (a, b, true) \\
\texttt{ln_i}(x, r+1) &= (a + (x \overset{d}{\uparrow} r+2) \overset{d}{\oslash} (r+2), \\
&\quad\; b + (x \overset{u}{\uparrow} r+2) \overset{u}{\oslash} (r+2), true) \\
&\quad\; \text{when} \quad \texttt{ln_i}(x, r) = (a, b, \mathit{false})
\end{aligned}$$

We prove quite easily in the domain of real numbers, using proofs by induction on r, that if x is positive and $\texttt{ln\_i}(x, r) = (a, b, c)$, then c is `true` exactly when r is even, and the following interval is satisfied:

$$a \leq \sum_{k=0}^{r} (-1)^k \frac{x^{k+1}}{k+1} \leq b$$

4.4 Second stage: with integers

We should now make explicit the links between functions on real numbers and integers. At this point, the difference between various proof systems is probably important. Our experiment was performed in the Coq system [13], where several types of numbers are in use. For this section, we will mostly consider the types of natural numbers, integers, and real numbers. The first two are inductive types, which means that they are algebraic datatypes, and they come with an intrinsic notion of computation. The type `nat` of natural numbers follows the structure of *Peano* axioms, so that representing a number n uses an amount of memory propotional to n. The type `Z` of integers follows the structure of sequences of bits, so that representing a number n uses an amount of memory proportional to $log(n)$. These types also come with pattern-matching and recursive computation, so that the time needed to add n with m is proportional to n for natural numbers and and proportional to $max(log(n), log(m)$ for integers.

The type of real numbers is not inductive, and it does not provide any means of computation. Addition, multiplication, and the other operations are only manipulated as abstract operations, with no meaning of computation. However, these operations are known through their properties. For instance, addition is associative and commutative, multiplication distributes and so on. Large integer constants can be written in Coq, but this is only a parsing and displaying trick: when the user types the number 11, the parser actually constructs the following expression:

```
1 + (1 + 1) * (1 + (1 + 1) * (1 + 1))
```

One can compute with these expressions quite fast, using the same algorithms as for sequences of bits[2], In Coq for instance, the tactic called "`psatzl R`" [4, 5] takes negligible time to verify the and accept the equality

$$\begin{aligned}
&12345678912345678912345678 \quad + \\
&87654321087654321087654321 \quad \leq \\
&99999999999999999999999999
\end{aligned}$$

but these algorithms only work for real numbers that represent integers, so there is no simple solution when one wants to compare, for instance $\sqrt{2} + \sqrt{3}$ with π. Such a comparison can be made and proved correct, but an ad hoc algorithm must be used.

[2] using the pattern `(1 + 1) * ...` as if it was a bit set to 0 and the pattern `1 + (1 + 1) * ...` as if it was a bit set to 1

There are bridges between the types `nat` and `Z` and the type `R`. Two functions `INR` and `IZR` take as input natural numbers and integers and produce the corresponding real number. In the other direction, there is a function from `R` to `Z` that maps any x in `R` to the smallest integer `n` in `Z` such that $x \leq \texttt{IZR}(n)$. Using this last function, *floor* and *ceiling* functions can be defined.

When it comes to representing real numbers with integers in a fixed point setting, we simply map integers to real numbers using the following function (`hR` is an acronym for *high-precision to Real*). In this development, we use `magnifier` as a name for the constant we wrote μ in the mathematical formulas.

```
hR(n) = IZR(n) / IZR(magnifier)
```

When it comes to writing the converse, we actually define two functions, which correspond to two rounding modes:

```
Rh(x) = floor (x * IZR(magnifier))
Rh'(x) = ceiling (x * IZR(magnifier))
```

Both `Rh` and `Rh'` are right inverse to `hR`, but in general, for any x that is not representable as a fixed precision number, $\texttt{hR}(\texttt{Rh}(x)) \neq x \neq \texttt{hR}(\texttt{Rh}'(x))$.

Thanks to its inductive nature, the datatype of integers is equiped with a collection of algorithms, including Euclidean division and square root. The last two actually return pairs of output, where one is the closest integer below the intended real value, and the second one is the remainder. To construct rounded up results of division, it is only necessary to compare this remainder with 0 and act accordingly.

In the end, we define functions as described in section 3 and we prove that they satisfy the properties described in section 4.3. To formulate these properties, we need to use the functions `hR`, `Rh`, and `Rh'`. Here is an example:

```
Lemma up_div_spec : forall x y, 0 <= x -> 1 <= y ->
      x / y <= hR (up_div (Rh' x) (Rh y)).
```

In this statement, `up_div` is a function where all inputs and outputs are integers. Algorithms for computing values eventually manipulate only integers and only use these functions. The proof of correctness for these algorithms only amounts to showing that the computations done on integers mirror exactly the computations done in the same algorithm using the function $\cdot \overset{u}{\oslash} \cdot$, etc. This could be very easy, but we have to thread the condition that all arguments to all functions are positive in the algorithm correctness proof.

4.5 Illustration: computing logarithm bounds

We applied the techniques of this first section to develop a function that computes lower and upper bounds for logarithms. The first step was to define a formal series to compute some logarithm values. This series is simply obtained by observing the properties of the inverse:

$$\frac{1}{1+y} = 1 - y + y^2 - y^3 + \cdots$$

More precisely, since the series is alternating, we have the following property

$$1-y+y^2-y^3\cdots-y^{2k+1}\frac{1}{1+y} << 1-y+y^2-y^3+\cdots+y^{2k}.$$

A well known property of the logarithm function is that its derivative is the inverse function, thus

$$y - \frac{y^2}{2} + \frac{y^3}{3}\cdots - \frac{y^{2k}}{2k} < \ln(1+y)y - \frac{y^2}{2} + \frac{y^3}{3}\cdots + \frac{y^{2k+1}}{2k+1}.$$

Substituting $x-1$ for y in these formulas, we obtain the following comparisons

$$\sum_{i=0}^{2k+1}(-1)^i\frac{x^{i+1}}{i+1} < \ln x < \sum_{i=0}^{2k}(-1)^i\frac{x^{i+1}}{i+1}.$$

These inequalities are true as soon as x is greater than or equal to 1. For x greater than 2, the series diverges, this means that the bounds provided by these equalities get worse and worse even as the order k increases.

Thus, to be able to compute logarithms for values outside the interval [1,2], one needs to apply another technique, known as *range reduction*. For numbers larger than 2 we chose to rely on square roots. For x in $[2,4]$, we know that $\sqrt{x}$ is in $[1,2]$ and we can use the series to compute $\ln\sqrt{x}$ and then conclude by using the following property:

$$\ln x = 2\ln\sqrt{x}.$$

Actually, it is also useful to apply this technique in the neighborhood of 2, because it leads to computing the power series at a value around 1.41, and this power series converges much better as one get closer to 1. We chose to implement a function that performs 4 steps of range reduction using square roots. Here is the Coq code for this function. The `( . )%Z` expresses that the two values returned in the pair are integers. These integers are bounds for $\ln n \times \mu$.

```
Definition ln_approx n p :=
 let v := Z.sqrt (Z.sqrt (Z.sqrt
             (Z.sqrt (n * magnifier ^ 15)))) in
 let '(_, b, _) :=
      ln_i (v + 1 - magnifier) (2 * p) in
 let '(a, _, _) :=
      ln_i (v - magnifier) (S (2 * p)) in
   (16 * a, 16 * b)%Z.
```

The correctness of this function is expressed by the following statement, which has been formally proved:

```
ln_approx_correct :
  forall x, (magnifier < x)%Z ->
  forall p, let (a, b) := ln_approx x p in
    hR a <= ln (hR x) <= hR b.
```

Notice that this statement provides bounds for $\ln x$, but it does not guarantee that these bounds improve when p increases. In practice, this function is useful only when $\texttt{hR}(x) < 64$, but this fact has not been formally proved.

4.6 Intermediate conclusion

In this section we basically described interval computations in a fixed point setting, where all computations are represented using integers. Interval computations mean that we do the computation twice, once for the lower bound and once for the upper bound, making sure to use lower and upper bounds of intermediate results appropriately. In the next section, we will take another approach, which makes it possible to perform the computation once, by computing bounds for the error. This is useful if we want to perform the computation at such a precision that each elementary operation is too costly to be duplicated.

5. Estimating bounds

Instead of computing lower and upper bounds for the results, we will now compute the errors brought by each approximation, and see how these errors fare in later computations. This will make it possible that sometimes errors compensate in such a way that they ultimately become negligible. This approach will rely on more mathematical properties of the functions being computed, especially the derivative.

5.1 A new running example

In their book about properties of the arithmetic-geometric mean [6], Borwein and Borwein use a sequence defined as follows:

$$\begin{aligned} y_0 &= \sqrt{2} \\ y_{n+1} &= \frac{y_n + 1}{2\sqrt{y_n}} \end{aligned}$$

This sequence converges towards 1, but it is used to construct another sequence that converges towards π very fast (the number of known digits doubles at every iteration). Because of this fast convergence, it is reasonable to use this sequence as the basis for an algorithm to compute large numbers of decimals of the π number (for instance, to compute one million digits, only twenty iterations are required), but all operations must be performed with the same precision. If this precision is really high, we want to avoid performing the same operation twice.

5.2 More precise properties of rounded down operations

In section 4.3, we only stated that the rounded down operations gave lower bounds to the result of their exact counterparts. Actually, we could have said something more precise: the distance with the exact operation is actually bounded by a precise value, $e = \mu^{-1}$. So we should actually use the following properties in our formal reasoning.

$$0 \leq a \wedge 0 \leq b \Rightarrow a \times b - e < a \overset{d}{\otimes} b \leq a \times b$$

$$0 \leq a \wedge 0 < b \Rightarrow \frac{a}{b} - e < a \overset{d}{\oslash} b \leq \frac{a}{b}$$

$$0 \leq a \Rightarrow \sqrt{a} - e < \sqrt[\sqcup]{a} \leq \sqrt{a}$$

Repeated use of these rounded operations do accumulate errors, of the order of magnitude of e at each step, but the errors obtained from previous operations are modified in a way that depends on the function actually being computed, and it is instrumental to take the global picture into account to see how this modification operates.

5.3 Error processing

It is well known that naive interval computation misses opportunities when applied at a level that is too local. The typical example is subtraction. If we know bounds for x and y, say $x \in [x_c - e, x_c + e]$ and $y \in [y_c - e, y_c + e]$, then we know that $x - y$ is an interval obtained by combining both bounds of each input:

$$x - y \in [x_c - y_c - 2e, x_c - y_c + 2e].$$

When the inputs are known with a possible error of e, the outputs are known with a possible error of $2e$.

If we apply this scheme to x and x, then we are computing $x - x$, and the result should be 0, exactly. There is no need for an error here. So the naive mechanism of computing bounds for output intervals from the input intervals is particularly inefficient in this case.

The solution we propose in this section is to avoid working with intervals, but instead to work with an explicit error, say h, and to study the difference between the result of our computation when applied to $x + h$ with the result of the exact mathematical value when computed with x.

Expressed in abstract terms, our problem is to compute some value $f(x)$, except that we don't know x exactly we only know a value $x + h$, where h is unknown but can be bounded, and we don't have a good implementation of f, we only have some other function $\boxed{\mathrm{f}}$, built using approximated elementary operations, which each had some unknown error which can also be bounded. So the value we are actually computing is $\boxed{\mathrm{f}}(x + h)$ instead of $f(x)$. Our approach will be to first study $\boxed{\mathrm{f}}(x + h) - f(x + h)$ and find a bound, and second to study $f(x + h) - f(x)$ and find a bound.

For instance, in the case of our new running example, we want to compute the distance between

$$\frac{1 + y}{2\sqrt{y}} \qquad \text{and} \qquad (1 + yh) \overset{d}{\oslash} (2 \sqrt[\sqcup]{y + h}).$$

Note that we already explained that addition and multiplication by an integer can be performed without adding errors in our setting.

The difference between the exact value and the computed value can be decomposed in two parts: one part comes from the errors of the rounded operations, the other part come from the processing in the input's explicit error, h.

5.4 Immediate rounding errors

In this section, we want to study the difference $\boxed{\mathrm{f}}(x+h) - f(x+h)$. This study can be analysed by following the structure of the computation, starting from the most external rounded operators.

In the case of our running example, we perform the study in two different steps, one for the upper bound and one for the lower bound.

$$\begin{aligned} (1 + y + h) \overset{d}{\oslash} (2 \sqrt[\sqcup]{y + h}) &\leq \frac{1 + y + h}{2 \sqrt[\sqcup]{(y + h)}} \\ &\leq \frac{1 + y + h}{2 \times (\sqrt{y + h} - e)} \\ &\leq \frac{1 + y + h}{2\sqrt{y + h}} + \frac{3}{2} \times e \end{aligned}$$

In the last step, we use several facts from our context, first y is taken in the interval $[1, \sqrt{2}]$, second e is positive and small, and third, $\frac{1}{1-e} < 1 + e + 2e^2$. So already in this example, we see that two rounded operations only add errors of the order of magnitude of three halves of the basic rounding error. These comes from the fact that we are only using rounded down operations, while we are looking at an upper bound of the result, so the division operation actually does not contribute to the rounding error.

A similar study on the other side gives the following lower bound. This time, the square root does not contribute to the rounding error.

$$\frac{1 + y + h}{2\sqrt{y + h}} - e \leq (1 + y + h) \overset{d}{\oslash} (2 \sqrt[\sqcup]{y + h})$$

5.5 Error propagation

We now need to study the difference $f(x + h) - f(x)$, since h is supposed to be an error, it is supposed to be small, and a first order approximation using the mean value theorem should suffice. This theorem states that if f is suitably continuous and derivable, then there exists some c between x and $x + h$ such that the difference is the derivative $f'(c)$ multiplied by the error h.

$$f(x + h) - f(x) = f'(c) \times h$$

At this stage, context information will be useful, to help keep the derivative $f'(c)$ small in absolute value.

In the case of our running example, this means that we need to compute the derivative of the function

$$y \mapsto \frac{1 + y}{2\sqrt{y}}.$$

This derivative is only valid when y is non-zero and it has the following value:

$$\frac{y - 1}{4y\sqrt{y}}.$$

If we know that y is between 1 and $\sqrt{2}$ and h is smaller than $1/10$, then we can guarantee that the derivative is smaller than $1/8$ in absolute value, therefore we have the following approximation information:

$$\left|\frac{1+y+h}{2\sqrt{y+h}} - \frac{1+y}{2\sqrt{y}}\right| < \frac{1}{8}|h|$$

We see that h is actually reduced in the process: the error coming from previous computations is actually small when compared to the error produced in the last elementary operations.

We can then wrap up the study of y_n in the following fashion: if division and square root operations are implemented with rounded down mode, with an error of at most e at each operation, and if y_n is known with an error does not exceed $2e$, then y_{n+1} is also computed with an error does not exceed $2e$ (because $3/2 + 1/8 < 2$. The intial value is itself computed using a rounded down square root, so the initial error does not exceed e, so we know that all elements of y_n will be computed with an error that does not exceed $2e$. If we want to compute the elements of this sequence with one million digit of precision, only one million and one digit for intermediate computations will be needed in most cases[3].

6. Using big integers

As formalized in the Coq system, addition and comparison should take a time that is proportional to the number of bits of one of the operands. Other operations rely naively on addition and comparisons, so that their complexity grows quite fast. For instance, here is a table comparing the times for comparing 3^n and 5^m, for different values of n and m (actually m is chosen so that the two compared numbers have the same order of magnitude).

n, m	100, 68	1000, 682	10000,6827	100000, 68261
time	0.002s	0.07s	5s	$>$ 5mn

Other researchers have been using Coq for intensive numerical proofs before [1, 10] and their library for big integers was integrated in the standard version. In particular, that library re-uses an algorithm for square roots that had been formally proved correct in [3]. For instance, the same computations using big integers produce the following table.

n, m	100, 68	1000, 682	10000,6827	100000,68261
time	0.002s	0.003s	0.03s	1.4s

To switch from the basic integer library to the big-integer library, it is enough to replace systematically all elementary operations on integers with elementary operations on big integers. For the correctness proof, we can simply rely on the direct correspondance between integers and big integers. This correspondance is given in the form of theorems with the following shape.

```
BigZ.spec_mul :
  forall x y : bigZ, [x * y] = ([x] * [y])%Z
```

In this theorem the notation `[ . ]` stands for the function that maps any big integer to the integer it represents. We simply need to push the correspondance through our algorithm, by systematic rewriting, thus obtaining obtaining similar theorems for each of our functions. The only difficulty comes from the functions that do not simply return an integer result, but rather a pair of an integer and a boolean value or a pair of two integers, because the use of these functions does not directly suit proofs based solely on rewriting.

In this big integer library, it is more efficient to compute with a large power of 2 than with a large power of 10. If the goal is to compute one million decimal digits, it is necessary to find the number n such that $10^{10^6} < 2^n$ and provide a proof of this comparison. This number[4] is simply

$$n = \left\lceil \frac{\ln 10}{\ln 2} \times 10^6. \right\rceil$$

This was the motivation for the first part of this paper. For smaller numbers, developing a dedicated proof to describe formally verified of the bounds is not necessary, it takes only one second to compare 10^{10^5} and the suitable power of 2, but the time is multiplied by 30 for 10^{10^6}.

Once we have chosen a suitable power of 2, we can measure the gain provided by the change from powers of 10 to powers of 2. Computing the integer square root of 2×2^{2n} takes $2/3$ of the time to compute the square root of $2^{2\times 10^6}$.

7. Application: computing π to one million digit

In addition to the sequence y_n, Borwein and Borwein [6] give the following sequences[5] that make it possible to compute good approximations of π.

$$\begin{aligned} z_1 &= \sqrt{\sqrt{2}} \\ z_{n+1} &= \frac{1+z_n y_n}{(1+z_n)\sqrt{y_n}} \\ \pi_n &= (2+\sqrt{2}) \prod_{k=1}^{n} \frac{1+y_k}{1+z_k} \\ b_n &= \frac{4\pi_0}{500^{2^n}} \end{aligned}$$

The convergence of π_n towards π is expressed by the following comparisons:

$$0 < \pi_{n+1} - \pi < b_n$$

The particular shape of the bound b_n expresses that the number of known digits should be multiplied by 2 at each iteration. The mathematical proofs for this result have been formally verified, but this is not the topic of this paper, where we are only concerned with the task of representing the real number of computations in our fixed point setting.

To obtain one million digits of π we must compute a few more digits, to allow for errors coming both from the rounding and errors and from $\pi_n - \pi$. Let's first evaluate the rounding errors.

We already showed that the y_n values can be computed with an error that does not exceed twice the rounding error of the elementary rounded operations $2 \times \mu^{-1}$. A similar result makes it possible to show that the accumulated error in z_n is only $4 \times \mu^{-1}$. The next step is to observe the computation of the big product

$$\prod_{k=1}^{n} \frac{1+y_k}{1+z_k}.$$

In the concrete computation, `prod`, all the inputs representing the values y_n and z_n are tainted, and moreover the products and divisions also introduce more errors. We were not able to prove the same kind of stability as for y_n and z_n, but we still get a good bound on the error:

$$|\texttt{prod}(n) - \prod_{k=1}^{n} \frac{1+y_k}{1+z_k}| < \frac{6n+2}{\mu}$$

Now, if we name $\texttt{rpi}(n)$ our concrete computation of π_n, we get the following bound on the error.

$$|\texttt{rpi}(n) - \pi_n| < \frac{21n+2}{\mu}$$

[3] this is tricky only if the millionth digit is 0, 1, 8, or 9.

[4] the right value for n is 3321929

[5] Actually, Borwein and Borwein use slightly different naming conventions.

To compute a million digits of π, we expect n to be around 20, so that the numerator is smaller than 500. Therefore we need to compute approximately 4 extra digits (or 14 extra bits) to help make sure that even the millionth digit is exact. Actually, we need a bit of luck here: even if we compute 4 extra digits and these digits give a number smaller than 400 or larger than 9600, we don't know if the millionth digit is given accurately. For this reason, we implemented a computation with $10^{10^6}+4$ digits, obtaining a number N and we then computed $N \bmod 10^4$ to check whether we are within the bounds[6].

If we want to compute one million digits of π, we need to compute the right n so that $b_n < 10^{-10^6}$. By computing $\ln 2$ and $\ln 10$ we know that $20 \ln 2 > 6 \ln 10$, therefore $2^{20} > 10^6$, and we can conclude using the fact that the logarithm function is strictly increasing.

$$\ln b_{19} = \ln(4\pi_0) - 2^{19} \ln 5 - 2^{20} \ln 10 \leq -2^{20} \ln 10 \leq -10^6 \ln 10$$

Performing this kind of proof using the logarithm function is more practical than computing directly the big constants and then their logarithms, because the intermediate computations concern values that are not practical. However, in early version of our work, before we implemented a computational version of the logarithm function, relied solely on removing power functions from the formulas to compare. Actually comparing 2^{20} and 10^6 is an easy computation that can be integrated directly in the proof.

To summarize the computations that are performed: we compute π_{20}, in binary format, actually computing 3321942 bits, and we then multiply the result by 10^{10^6+4} and divide by $2^{3321942}$ to obtain an approximation of $\pi_{20} \times 10^{10^6+4}$ with a possible error less than 1. This computation requires computing twenty high precision square roots, sixty divisions, and forty multiplications. The final result is divided by 10^4, we check that the remainder is larger than 427 and smaller than 9573. This is enough to guarantee that we computed exactly one million digits of π.

The Coq code describing this process is summarized in the following definition and the lemma `pi_osix` expresses that this definition is correct.

```
Definition million_digit_pi :=
let magnifier := (2 ^ 3321942)%bigZ in
let n := hpi magnifier 20 in
let n' := (n * 10 ^ (10 ^ 6 + 4) /
           2 ^ 3321942)%bigZ in
let (q, r) := BigZ.div_eucl n' (10 ^ 4) in
(((427 <? r)%bigZ && (r <? 9573)%bigZ)%bool, q).

pi_osix :
 fst million_digit_pi = true ->
   hR (10 ^ (10 ^ 6)) (snd million_digit_pi)
      < PI <
   hR (10 ^ (10 ^ 6)) (snd million_digit_pi) +
   Rpower 10 (-(Rpower 10 6)).
```

We ran this code on a powerful computer in two different setting. The first setting relies on *virtual machine computation*, as designed by B. Grgoire and available in the standard version of Coq 8.4. In this case, execution takes about 6 hours and a half. The second setting relies on *native computation* as designed by M. Dénès[7] In this case, execution takes approximately forty minutes, not including the time taken by the Coq system to display the result (about 8 more minutes). The display routines are not part of the code that was proved formally.

[6] and after checking, we know this is satisfied.

[7] 40 minutes with 63 bit-integers (version provided by Dénès); an hour and a half with 31 bit-integers (development version of Coq at the time of writing).

References

[1] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending coq with imperative features and its application to SAT verification. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010*, volume 6172 of *LNCS*, pages 83–98. Springer, 2010.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[3] Yves Bertot, Nicolas Magaud, and Paul Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29:225–252, 2002.

[4] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *LNCS*, pages 48–62. Springer, 2006.

[5] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT proofs for fast reflexive checking inside coq. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP*, volume 7086 of *LNCS*, pages 151–166. Springer, 2011.

[6] Jonathan M. Borwein and Peter B. Borwein. *Pi and the AGM*. Wiley Interscience, 1987.

[7] Richard P. Brent. Fast multiple-precision evaluation of elementary functions. *Journal of the Association of Computation Machinery*, 23:242–251, 1976.

[8] Marc Daumas, David R. Lester, and César Muñoz. Verified real number calculations: A library for interval arithmetic. *IEEE Trans. Computers*, 58(2):226–237, 2009.

[9] Marc Daumas, Guillaume Melquiond, and César Muñoz. Guaranteed Proofs Using Interval Arithmetic. In *17th IEEE Symposium on Computer Arithmetic*, pages 188–195, Cape Cod, Massachusetts, États-Unis, 2005. IEEE.

[10] Benjamin Grégoire and Laurent Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *LNCS*, pages 423–437. Springer, 2006.

[11] John Harrison. The HOL Light theorem prover, 2010.

[12] Cezary Kaliszyk and Russel O'Connor. Computing with Classical Real Numbers. *Journal of Formalized Reasoning*, 2(1), 2009.

[13] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2012. Version 8.4.

[14] Guillaume Melquiond. Floating-point arithmetic in the Coq system. In *8th Conference on Real Numbers and Computers*, pages 93–102, Santiago de Compostela, Spain, 2008.

[15] Eugene Salamin. Computation of π Using Arithmetic-Geometric Mean. *Mathematics of Computation*, 30(135):565–570, July 1976.

[16] Alexey Solovyev. *Formal Computations and Methods*. PhD thesis, University of Pittsburgh, 2012.

[17] Alexey Solovyev and Thomas C. Hales. Formal verification of nonlinear inequalities with taylor interval approximations. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013*, volume 7871 of *LNCS*, pages 383–397. Springer, 2013.

[18] Paul Zimmermann. Reliable computing with GNU MPFR. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Third International Congress on Mathematical Software, ICMS 2010*, volume 6327 of *LNCS*, pages 42–45. Springer, 2010.