Formal verification of floating point trigonometric functions

John Harrison

Intel Corporation, EY2-03
5200 NE Elam Young Parkway
Hillsboro, OR 97124, USA

Abstract. We have formal verified a number of algorithms for evaluating transcendental functions in double-extended precision floating point arithmetic in the Intel® IA-64 architecture. These algorithms are used in the Itanium™ processor to provide compatibility with IA-32 (x86) hardware transcendentals, and similar ones are used in mathematical software libraries. In this paper we describe in some depth the formal verification of the *sin* and *cos* functions, including the initial range reduction step. This illustrates the different facets of verification in this field, covering both pure mathematics and the detailed analysis of floating point rounding.

1 Introduction

Code for evaluating the common transcendental functions (*sin*, *exp*, *log* etc.) in floating point arithmetic is important in many fields of scientific computing. For simplicity and portability, various standard algorithms coded in C are commonly used, e.g. those distributed as part of FDLIBM (Freely Distributable LIBM).[1] However, to achieve better speed and accuracy, Intel is developing its own mathematical algorithms for the new IA-64 architecture, hand-coded in assembly language.

Many of these algorithms are complex and their error analysis is intricate. Subtler algorithmic errors are difficult to discover by testing. In order to provide improved quality assurance, Intel is subjecting the most important algorithms to formal verification. In particular, the IA-64 architecture provides full compatibility with IA-32 (x86), and the latter includes a small set of special instructions to evaluate core transcendental functions. These will be the focus here, though essentially the same techniques can be used in other contexts.

Many of the algorithms for evaluating the transcendental functions follow a similar pattern. However, IA-64 supports several floating-point precisions: single precision (24 significand bits), double precision (53) and double-extended precision (64), and the target precision significantly affects the design choices. Since internal computations can be performed in double-extended precision, rounding

[1] See `http://www.netlib.org/fdlibm`.

errors are much less a concern when the overall computation is for single or double precision. It is relatively easy to design simple, fast and accurate algorithms of the sort Intel provides [11]. For double-extended precision functions — such as the IA-32 hardware transcendentals — much more care and subtlety is required in the design [15] and the formal verifications are significantly more difficult.

In the present paper, to avoid repetition and dilution, we focus on the formal verification of an algorithm for a particular pair of functions: the double-extended floating point sine and cosine. This is used for compatibility with the IA-32 hardware transcendentals `FSIN`, `FCOS` and `FSINCOS`. Essentially the same algorithm, with the option of a more powerful initial range-reduction step for huge input arguments, is used in Intel's double-extended precision mathematical library that can be called from C and FORTRAN. These particular functions were chosen because they illustrate well the many aspects of the formal verifications, involving as they do a sophisticated range reduction step followed by a tricky computation carefully designed to minimize rounding error. They are somewhat atypical in that they do not use a table lookup [16], but otherwise seem to show off most of the interesting features.

2 Outline of the algorithm

The algorithm is intended to provided accurate double-extended approximations for $sin(x)$ and $cos(x)$ where x is a double-extended floating point number in the range $-2^{63} \leq x \leq 2^{63}$. (Although there are separate entry points for sine and cosine, most of the code is shared and both sine and cosine can be delivered in parallel, as indeed is required by `FSINCOS`.) According to the IA-32 documentation, the hardware functions just return the input argument and set a flag when the input is out of this range. The versions in the mathematical library, however, will reduce even larger arguments. An assembler code implementation of the mathematical library version is available on the Web, and this actual code can be examined as a complement to the more abstract algorithmic description given here.

`http://developer.intel.com/software/opensource/numerics/index.htm`

The algorithm is separated into two phases: an initial range reduction, and the core function evaluation. Mathematically speaking, for any real number x we can always write:

$$x = N(\pi/2) + r$$

where N is an integer (the closest to $x \cdot \frac{2}{\pi}$) and $|r| \leq \pi/4$. We can then evaluate $sin(x)$ and/or $cos(x)$ as either $sin(r)$, $cos(r)$, $-sin(r)$ or $-cos(r)$ depending on N modulo 4. For example:

$$sin((4M + 3)(\pi/2) + r) = -cos(r)$$

We refer to the process of finding the appropriate r and N (modulo 4 at least) as *trigonometric range reduction*. The second phase, the core evaluation, need now only be concerned with evaluating $sin(r)$ or $cos(r)$ for r in a limited range, and then perhaps negating the result, depending on N modulo 4. Moreover, since $cos(x) = sin(x + \pi/2)$ we can perform the same range reduction and core evaluation for $sin(x)$ and $cos(x)$, merely adding 1 to N at an intermediate stage if $cos(x)$ is required. Thus, as hinted earlier, the sine and cosine functions share code almost completely, merely setting an initial value of N to be 0 or 1 respectively.

The core evaluation of $sin(r)$ and $cos(r)$ can now be performed using a power series expansion similar to a truncation of the familiar Taylor series:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$
$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

but with the pre-stored coefficients computed numerically to minimize the maximum error over r's range, using the so-called *Remez algorithm* [13]. The actual evaluations of the truncated power series in floating point arithmetic, however, require some care if unacceptably high rounding errors are to be avoided.

3 HOL floating point theory

The verification described here is conducted in the HOL Light theorem prover [7], and the formal proofs are founded on formalized HOL theories of mathematical analysis [9] and floating point arithmetic [10]. Because of space limitations, we cannot describe these theories in great detail here, but we will sketch a few highlights, particularly of the floating point material where there is less established notation. We hope this will suffice for the reader to follow the explicit HOL theorems given below.

HOL Light is a highly foundational theorem proving system using the methodology first established by Edinburgh LCF [5]. 'LCF style' provers explicitly generate proofs in terms of extremely low-level primitive inferences, in order to provide a high level of assurance that the proofs are valid. In HOL Light, as in most other LCF-style provers, the proofs (which can be very large) are not usually stored permanently, but the strict reduction to primitive inferences is maintained by the abstract type system of the interaction and implementation language, which for HOL Light is CAML Light [3, 17]. The primitive inference rules of HOL Light, which implements a simply typed classical higher order logic, are very simple. However CAML Light also serves as a programming medium allowing higher-level derived rules (e.g. to automate linear arithmetic, first order logic or reasoning in other special domains) to be programmed as automatic reductions to primitive inferences. This lets the user conduct the proof at a more palatable level, while still maintaining the logical safety that comes from

low-level proof generation. A few application-specific instances of programming derived rules in HOL Light will be described in the present paper.

HOL's foundational style is also reflected in its approach to developing new mathematical theories. All HOL mathematics is developed by *constructing* new structures from the primitive logical and set theoretic basis, rather than by asserting additional axioms. For example, the natural numbers are constructed by inductively carving out an appropriate subset of the infinite set asserted to exist by the basic Axiom of Infinity. In turn, such inductive definitions are defined as appropriate set-theoretic intersections, rather than being permitted as primitive extensions. The positive real numbers are defined as equivalence classes of nearly-additive functions $\mathbb{N} \to \mathbb{N}$ (equivalent to a version of Cantor's construction using Cauchy sequences, but without explicit use of rationals), and reals as equivalence classes of pairs of positive real numbers. After the development of some classical analytical theories of limits, differentiation, integration, infinite series etc., the most basic transcendental functions (*exp*, *sin* and *cos*) are defined by their power series and proved to have their standard properties. Some 'inverse' transcendental functions like *ln* and *atn* are defined abstractly and their properties proven via the inverse function theorem. For more details, the reader can consult [9].

HOL notation is generally close to traditional logical and mathematical notation. However, the type system distinguishes natural numbers and real numbers, and maps between them by `&`; hence `&2` is the real number 2. The multiplicative inverse x^{-1} is written `inv(x)` and the power x^n as `x pow n`. Note that the expression `Sum(m,n) f` denotes $\Sigma_{i=m}^{m+n-1} f(i)$, and not as one might guess $\Sigma_{i=m}^{n} f(i)$.

Much of the theory of floating point numbers is generic. Floating point formats are identified by triples of natural numbers `fmt` and the corresponding set of representable real numbers, ignoring the upper limit on the exponent range, is `iformat fmt`. The second field of the triple, extracted by the function `precision`, is the precision, i.e. the number of significand bits. The third field, extracted by the `ulpscale` function, is N where 2^{-N} is the smallest nonzero floating point number of the format.

Floating-point rounding is performed by `round fmt rc x` which denotes the result of rounding the real number `x` into `iformat fmt` (the representable numbers, with unlimited exponent range, of a floating point format `fmt`) under rounding mode `rc`. The predicate `normalizes` determines whether a real number is within the range of normal floating point numbers in a particular format, i.e. those representable with a leading 1 in the significand, while `losing` determines whether a real number will lose precision, i.e. underflow, when rounded to a given format.

Most of the difficulty of analyzing floating point algorithms arises from the error committed in rounding floating point results to fit in their destination floating point format. The theorems we use to analyze these errors can be subdivided into (i) routine worst-case error bound theorems, and (ii) special cases where no rounding error is committed.

3.1 The $(1 + \epsilon)$ property

In typical error analyses of high-level floating point code, the first kind of theorem is used almost exclusively. The mainstay of traditional error analysis, often called the '$(1+\epsilon)$' property, is simply that the result of a floating point operation is the exact result, perturbed by a relative error of bounded magnitude. Recalling that in our IEEE arithmetic, the result of an operation is the rounded exact value, this amounts to saying that x rounded is always of the form $x(1 + \epsilon)$ with $|\epsilon|$ bounded by a known value, typically 2^{-p} where p is the precision of the floating point format. We can derive a result of this form fairly easily, though we need sideconditions to exclude the possibility of underflow (not overflow, which we consider separately from rounding). The main theorem is as follows:

```
|- ¬(losing fmt rc x) ∧ ¬(precision fmt = 0)
   ⟹ ∃e. abs(e) <= mu rc / &2 pow (precision fmt - 1) ∧
          (round fmt rc x = x * (&1 + e))
```

This essentially states exactly the '$1+\epsilon$' property, and the bound on ϵ depends on the rounding mode, according to the following auxiliary definition of `mu`:

```
|- (mu Nearest = &1 / &2) ∧ (mu Down = &1) ∧
   (mu Up = &1) ∧ (mu Zero = &1)
```

The theorem has two sideconditions, the second being that the floating point format is not trivial (it has a nonzero number of fraction bits), and the first being an assertion that the value x does not *lose precision*, in other words, that the result of rounding x would not change if the lower exponent range were extended. We will not show the formal definition [10] here, since it is rather complicated. However, a simple and usually adequate sufficient condition is that the exact result lies in the normal range or is zero:

```
|- normalizes fmt x ⟹ ¬(losing fmt rc x)
```

where

```
|- normalizes fmt x =
       (x = &0) ∨
       &2 pow (precision fmt - 1) / &2 pow (ulpscale fmt) <= abs(x)
```

There is also a similar theorem for absolute rather than relative error analysis, which is sometimes useful for obtaining sharper error bounds, but will not be shown here. It does not require any normalization hypotheses, which can make it simpler to apply.

3.2 Cancellation theorems

In lower-level algorithms like the ones considered here and others that the present author is concerned with verifying, a number of additional properties of floating point arithmetic are sometimes exploited by the algorithm designer and proofs of them are required for verifications. In particular, there are important situations where floating point arithmetic is exact, i.e. results round to themselves. This happens if and only if the result is representable as a floating point number:

```
|- a ∈ iformat fmt ⟹ (round fmt rc a = a)

|- ¬(precision fmt = 0) ⟹ ((round fmt rc x = x) = x ∈ iformat fmt)
```

There are a number of situations where arithmetic operations are exact. Perhaps the best-known instance is subtraction of nearby quantities; cf. Theorem 4.3.1 of [14]:

```
|- a ∈ iformat fmt ∧ b ∈ iformat fmt ∧ a / &2 <= b ∧ b <= &2 * a
   ⟹ (b - a) ∈ iformat fmt
```

Another classic result [12, 4] shows that we can obtain the sum of two floating point numbers exactly in two parts, one a rounding error in the other, by performing the floating point addition then subtracting both summands from the result, the larger one first:

```
|- x ∈ iformat fmt ∧
   y ∈ iformat fmt ∧
   abs(x) <= abs(y)
   ⟹ (round fmt Nearest (x + y) - y) ∈ iformat fmt ∧
      (round fmt Nearest (x + y) - (x + y)) ∈ iformat fmt
```

As we will see later, theorems of this sort, including some rather *ad hoc* derivative lemmas, are used extensively in the analysis of trigonometric range reduction, where almost every floating point operation is based on some special trick to avoid rounding error or later compensate for it! Some of these results are not readily found in the literature, but are well-known to experts in floating point arithmetic. Sometimes the lemmas we have ended up proving are not optimal and could profitably be sharpened, but having performed quite a few verifications we are confident that we have a fairly comprehensive basic toolkit.

4 Verification of range reduction

The principal difficulty of implementing trigonometric range reduction is that the input argument x may be large and yet the reduced argument r very small, because x is unusually close to a multiple of $\pi/2$. In such cases, the computation of r needs to be performed very carefully. Assuming we have calculated N, we need to evaluate:

$$r = x - N\frac{\pi}{2}$$

However, $\frac{\pi}{2}$ is irrational and so cannot be represented exactly by any finite sum of floating point numbers. So however the above is computed, it must in fact calculate

$$r' = x - NP$$

for some approximation $P = \frac{\pi}{2} + \epsilon$. The relative error $\frac{|r'-r|}{|r|}$ is then $\frac{N|\epsilon|}{|r|}$ which is of the order $|\frac{x\epsilon}{r}|$. Therefore, to keep this relative error within acceptable bounds (say 2^{-70}) the accuracy required in the approximation P depends on how small the (true) reduced argument can be relative to the input argument. In order to formally verify the accuracy of the algorithm, we need to answer the purely mathematical question: how close can a double-extended precision floating point number be to an integer multiple of $\frac{\pi}{2}$? Having done that, we can proceed with the verification of the actual computation of the reduced argument in floating point arithmetic.

4.1 Approximating π

For the proofs that follow we need to have an accurate rational approximation to π, and of course a *formal proof* that it is sufficiently accurate. The accuracy we need (2^{-225}) follows from the later proofs, but we first wrote a routine to approximate π to arbitrary precision, since less accurate approximations are useful for disposing of trivial sideconditions that crop up in proofs, e.g. $1 < \pi/2$.

Our starting point for approximating π is the Taylor series for the arctangent, which we had already derived for use in the verification of floating point arctangent functions. The proof of this proceeds as follows, using some of the real analysis described in [9]. We demonstrate, using the comparison test and the pre-proved convergence of the geometric series $\Sigma_{m=0}^{\infty} x^m$ for $|x| < 1$, that the infinite series $\Sigma_{m=0}^{\infty} \frac{(-1)^m}{2m+1} x^{2m+1}$ converges for $|x| < 1$ to some limit function $f(x)$. Therefore the series can provably be differentiated term-by-term, and the derivative series is $\Sigma_{m=0}^{\infty}(-1)^m(x^2)^m$ which is again a geometric series and sums to $\frac{1}{1+x^2}$. Consequently $f'(x)$ and $tan'(x) = \frac{1}{1+x^2}$ coincide for $|x| < 1$, and so $atn(x) - f(x)$ is constant there. But for $x = 0$ we have $atn(x) - f(x) = 0$, and hence it follows that the series converges to $atn(x)$ for all $|x| < 1$. The error in truncating the series can trivially be bounded provided $|x| \leq 1/2^k$ for $k > 0$, giving us the final theorem which in HOL looks like this:

```
|- abs(x) <= inv(&2 pow k) ∧ ¬(k = 0)
   ⟹ abs(atn x -
          Sum(0,n) (λm. (if EVEN m then &0
                         else --(&1) pow ((m - 1) DIV 2) / &m) *
                         x pow m))
       <= inv(&2 pow (n * k - 1))
```

It is now easy to obtain approximations to π by applying atn to appropriate rational numbers. We wrote a HOL derived rule `MACHIN_RULE` that computes (with proofs) linear forms in arctangents of rational numbers using the addition formula:

```
|- abs(x * y) < &1 ⟹ (atn(x) + atn(y) = atn((x + y) / (&1 - x * y)))
```

The user can apply `MACHIN_RULE` to any linear form in arctangents, and it is automatically decomposed into sums and negations, and the above theorem used repeatedly to simplify it, with the side-condition $|xy| < 1$ discharged automatically at each stage. In this way, useful approximation theorems that are variants on the classic Machin formula can be derived automatically without the user providing separate proofs:

```
#let MACHIN_1 = MACHIN_RULE `&4 * atn(&1 / &5) - atn(&1 / &239)`;;
MACHIN_1 : thm = |- pi / &4 = &4 * atn (&1 / &5) - atn (&1 / &239)
#let STRASSNITZKY_MACHIN = MACHIN_RULE
  `atn(&1 / &2) + atn (&1 / &5) + atn(&1 / &8)`;;
STRASSNITZKY_MACHIN : thm =
 |- pi / &4 = atn (&1 / &2) + atn (&1 / &5) + atn (&1 / &8)
#let MACHINLIKE_1 = MACHIN_RULE
   `&6 * atn(&1 / &8) + &2 * atn(&1 / &57) + atn(&1 / &239)`;;
MACHINLIKE_1 : thm =
 |- pi / &4 = &6 * atn (&1 / &8) + &2 * atn (&1 / &57) + atn (&1 / &239)
```

We use the last of these to derive approximations to π, again via a derived rule that for any given accuracy returns an approximation of π good to that accuracy. For example:

```
#let pth = PI_APPROX_RULE 5;;
pth : thm = |- abs (pi - &26696452523 / &8498136384) <= inv (&2 pow 5)
```

The above approach is easy to prove and program up, and adequate for the accuracies we needed for this proof, but for more precise approximations to π, we would probably need to exploit more efficient approximation methods for π such as the remarkable series [1] which we have already formally verified in HOL:

$$\pi = \Sigma_{m=0}^{\infty} \frac{1}{16^m}\left(\frac{4}{8m+1} - \frac{2}{8m+4} - \frac{1}{8m+5} - \frac{1}{8m+6}\right)$$

4.2 Bounding the reduced argument

Armed with the ability to find arbitrarily good rational approximations to π, we can now tackle the problem: how close can our input number be to an nonzero integer multiple of $\pi/2$? Every double-extended precision floating point number x with $|x| < 2^{64}$ can be written

$$x = k/2^e$$

for some integers $2^{63} \leq k < 2^{64}$ and $e > 0$; assuming the number x is normalized, k is simply its significand considered as an integer. We are then interested in bounding:

$$\begin{aligned} & |k/2^e - N\pi/2| \\ = & \frac{|N|}{2^e}(k/N - 2^e\pi/2) \end{aligned}$$

We only need to consider cases where $k/2^e \approx N\pi/2$ since otherwise x is not close to a multiple of $\pi/2$. So we can consider only N such that $|N| \leq 2^{65-e}/3.14159$. Moreover, we need only consider $e < 64$ since otherwise $|x| < 1$ and so x is too small to be close to a nonzero multiple of $\pi/2$. So, for each $e = 0, 1, \ldots, 63$ we just need to find the closest rational number p/q to $2^e\pi/2$ with $|q| \leq 2^{65-e}/3.14159$. We can then get a reasonable lower bound for $|k/2^e - N\pi/2|$ by:

$$\frac{2^{63-2e}}{3.1416}|p/q - 2^e\pi/2|$$

This is not quite optimal: we could use our knowledge of p and q to avoid the overestimate contained in the factor on the left, and rely on the fact that the term on the right would be significantly larger again for other p'/q'. However it is good to within a factor of 2, enough for our purposes.

We now have merely to solve (63 instances of) a classic problem of diophantine approximation: how close can a particular real number x be approximated by rational numbers p/q subject to some bound on the size of q? There is a well-established method for solving this problem, which is easy to formalize in HOL. Suppose we have two straddling rational numbers $p_1/q_1 < x < p_2/q_2$ such that $p_2q_1 = p_1q_2 + 1$. It is easy to show that any other rational approximation a/b with $b < q_1 + q_2$ is no better than the closer of p_1/q_1 and p_2/q_2. In such a case, unless p_1/q_1 and a/b are the same rational we must have $|p_1/q_1 - a/b| = |p_1b - q_1a|/(q_1b) \geq 1/(q_1b)$, since the numerator $p_1b - q_1a$ is a nonzero integer. Similarly, $|p_2/q_2 - a/b| \geq 1/(q_2b)$ unless $p_2/q_2 = a/b$. Therefore, since $|b| < q_1 + q_2$:

$$\begin{aligned} |p_1/q_1 - a/b| + |p_2/q_2 - a/b| &\geq 1/(q_1b) + 1/(q_2b) \\ &> 1/(q_1q_2) \\ &= |p_1/q_1 - p_2/q_2| \end{aligned}$$

Consequently a/b cannot lie inside the straddling interval, and so cannot be closer to x. This is easily proved in HOL:

```
|- (p2 * q1 = p1 * q2 + 1) ∧ ¬(q1 = 0) ∧ ¬(q2 = 0) ∧
   (&p1 / &q1 < x ∧ x < &p2 / &q2)
   ⟹ ∀a b. ¬(b = 0) ∧ b < q1 + q2
            ⟹ abs(&a / &b - x) >= abs(&p1 / &q1 - x) ∨
               abs(&a / &b - x) >= abs(&p2 / &q2 - x)
```

It remains to find these special straddling pairs of rational numbers, for it is not immediately obvious that they must exist. Note that the finding process does not need to produce any kind of proof; the numbers can be found via an arbitrary method and the property checked formally by plugging the numbers into the above theorem. The most popular method for finding such 'convergents' uses continued fractions [2]. We use instead a procedure that is in general less efficient but is simpler to program in our context, creating convergents iteratively by calculating the *mediant* of two fractions.

If we have two fractions p_1/q_1 and p_2/q_2 with $p_2q_1 = p_1q_2 + 1$ (and hence $p_1/q_1 < p_2/q_2$) then it is easy to prove that the mediant p/q where $p = p_1+p_2$ and $q = q_1 + q_2$ has the same property with respect to its parents: $pq_1 = p_1q + 1$ and $p_2q = pq_2 + 1$. Note that this implies $p_1/q_1p < q < p_2/q_2$ and that p/q is already in its lowest terms (any common factor of p and q would, since $p_2q = pq_2 + 1$, divide 1). In fact, iterating this generative procedure starting with just $0/1$ and $1/1$ generates all rational numbers between 0 and 1 in their lowest terms; this can be presented as the *Farey sequence* or *Stern-Brocot tree* [6].

We can now easily generate convergents to any real number x by binary chop: if we have $p_1/q_1 < x < p_2/q_2$ with $p_2q_1 = p_1q_2 + 1$, we simply form the mediant fraction p/q and iterate either with p_1/q_1 and p/q or with p/q and p_2/q_2 depending which side of the mediant x lies. It's easy to resolve this inequality via a sufficiently good rational approximation to x. We proceed until $q_1 + q_2$ reaches the bound we are interested in, and then plug the values into the main theorem, obtaining a lower bound on the quality of rational approximations to x. Finally, we use the very good rational approximation to π to get a good rational lower bound for the terms $p_1/q_1 - x$ and $p_2/q_2 - x$ in the conclusion. Iterating in this way for the various choices of e, we find our overall bound for how close the input number can come to a multiple of $\pi/2$: about $113/2^{76}$:

```
|- integer(N) ∧ ¬(N = &0) ∧
   a ∈ iformat (rformat Register) ∧ abs(a) < &2 pow 64
   ⟹ abs (a - N * pi / &2) >= &113 / &2 pow 76
```

4.3 Analyzing the reduced argument computation

The above theorem shows that, unless $N = 0$ in which case reduction is trivial, the reduced argument has magnitude at least around 2^{-69}. Assuming the input has size $\leq 2^{63}$, this means that an error of ϵ in the approximation of $\pi/2$ can constitute approximately a $2^{132}\epsilon$ relative error in r. Consequently, to keep the relative error down to about 2^{-70} we need $|\epsilon| < 2^{-202}$. Since a floating-point number has only 64 bits of precision, it would seem that we would need to approximate $\pi/2$ by four floating-point numbers $P_1, \ldots, P_4$ and face considerable complications in keeping down the rounding error in computing $x - N(P_1 + P_2 + P_3 + P_4)$. However, using an ingenious technique called *pre-reduction* [15], the difficulties can be reduced. Certain floating point numbers are exceptionally close to exact multiples of 2π; in fact slight variants of the methods used in

the previous section can easily find such numbers. In the present algorithms, a particular floating point number P_0 is used with

$$|P_0 - 4178442\pi| < 2^{-63.5}$$

(not, incidentally, so very far from our lower bound). By initially subtracting off a multiple of P_0, incurring a small error compared with subtracting off the corresponding even multiple of π, we then only need to deal with numbers of size $< 2^{24}$ in the main code. (The accurate subtraction of multiples of P_0 is similar in spirit to the main computation we discuss below, and we will not discuss it here for reasons of space.) Therefore, even in the worst case, we can store a sufficiently accurate $\pi/2$ as the sum of three floating-point numbers $P_1 + P_2 + P_3$, where the magnitude of P_{n+1} is less that half the least significant bit of P_n.

As noted earlier, the actual computations in the trigonometric range reduction rely heavily on special tricks to avoid or compensate for rounding error. The computation starts by calculating $y = P'x$, where $P' \approx \frac{2}{\pi}$, and then rounding it to the nearest integer N. Because P' is not exactly $\frac{2}{\pi}$, and the computation of y commits a rounding error, N may not be the integer closest to $x\frac{2}{\pi}$. However, it is always sufficiently close that the next computation, $s = x - NP_1$ (computed using a fused multiply-accumulate instruction, rather than by a separate multiplication and subtraction) is exact, i.e. no rounding error is committed. We will not show the rather messy general theorem that we use to justify this.

However, because P_1 is not exactly $\pi/2$, s is not yet an adequate reduced argument. Besides, we must deliver the reduced argument in two pieces $r + c$ with $|c| \ll |r|$, since in general merely coercing the reduced argument into a single floating point number would introduce unacceptable errors.

Different paths are now taken, depending on whether $|s| < 2^{-33}$. The path where this is not the case is simpler, so we will discuss that; the other uses the same methods but is significantly more complicated. Since $|s| \geq 2^{-33}$, it is sufficiently accurate to use $P_1 + P_2$, and hence all we need to do is subtract NP_2 from s. However, we need to make sure that $|c| \ll |r|$ for the reduced argument $r+c$ (the later computation would otherwise be inaccurate, since it neglects high powers of c in the power series), so we can't simply take $r = s$ and $c = -NP_2$. Instead we calculate:

$$\begin{aligned} w &= NP_2 \\ r &= s - w \\ c1 &= s - r \\ c &= c1 - w \end{aligned}$$

Apart from the signs (which are inconsequential, since rounding to nearest commutes with negation), the exactness of the all but the first line can be justified by the previous theorem on computing an exact sum, based on the easily proven fact that $|w| \leq 2^{-33} \leq |s|$. We have $r + c = s - w$ exactly, and so the only errors in the reduced argument are the rounding error in w and N times

the error in approximating $\pi/2$ by $P_1 + P_2$, both of which are small enough for our purposes.

5 Verification of the core computation

The core computation is simply a polynomial in the reduced argument. If the reduced argument is sufficiently small, very short and quickly evaluated polynomials suffice. (Note that this tends to even out the overall runtime of the algorithm, since it is exactly in the cases of small reduced arguments that the range reduction phase is more expensive.) In the extreme case where $|r| < 2^{-33}$, we can evaluate $cos(r + c)$ by just $1 - 2^{-67}$, regardless of the value of r. (In round-to-nearest mode this always rounds to 1, so the computation looks pointless, but we need to give sensible results when rounding down or towards zero). In the most general case, when we know only that $|r|$ is bounded by about $\pi/4$, the polynomials needed are significantly longer. For example the most general sin polynomial used is of the form:

$$p(y) = y + P_1 y^3 + P_2 y^5 + \cdots + P_8 y^{17}$$

where $y = r + c$ is the reduced argument, and the P_i are all floating point numbers. Note that the P_i are not the same as the coefficients of the familiar Taylor series (which in any case are not exactly representable as floating point numbers), but arrived at using the Remez algorithm to minimize the worst-case error over the possible reduced argument range. Evaluating all the terms with $y = r + c$ is unnecessarily complicated, since higher powers of c are negligible; instead the algorithm simply exploits the addition formula

$$sin(r + c) = sin(r)cos(c) + cos(r)sin(c) \approx sin(r) + c(1 - r^2/2)$$

and evaluates:

$$r + P_1 r^3 + P_2 r^5 + \cdots + P_8 r^{17} + c(1 - r^2/2)$$

The overall error, apart from that in range reduction which in this case where the reduced argument is not small is almost negligible, is now composed of three components:

- The approximation error $|p(r + c) - sin(r + c)|$.
- The additional error in neglecting higher powers of c: $|p(r+c)-(p(r)+c(1-r^2/2))|$.
- The rounding error in actually computing $p(r) + c(1 - r^2/2))$.

It is straightforward to provably bound the second error using some basic real algebra and analysis. The approximation and rounding errors are more challenging.

5.1 Bounding the approximation error

We have implemented an automatic HOL derived rule to provably bound the error in approximating a mathematical function by a polynomial over a given interval. The user need only provide a derived rule to produce arbitrarily good Taylor series approximations over that interval. For example, for the *cos* function, we can easily derive the basic Taylor theorem:

```
|- abs(x) <= inv(&2 pow k)
   ⟹ abs(cos x -
            Sum(0,n) (λm. (if EVEN m
                           then -- &1 pow (m DIV 2) / &(FACT m)
                           else &0) * x pow m))
       <= inv(&(FACT n) * &2 pow (n * k))
```

It's then straightforward to package this up as a derived rule, which we call `MCLAURIN_COS_POLY_RULE`. Given natural numbers k and p, this will compute the required n, instantiate the theorem and produce, with a proof, a polynomial $p(x)$ such that:

$$\forall x.\, |x| \leq 2^{-k} \Longrightarrow |cos(x) - p(x)| \leq 2^{-p}$$

For example:

```
#MCLAURIN_COS_POLY_RULE 3 7;;
it : thm =
 |- ∀x. abs x <= inv (&2 pow 3)
        ⟹ abs (cos x - poly [&1] x) <= inv (&2 pow 7)
#MCLAURIN_COS_POLY_RULE 2 35;;
it : thm =
 |- ∀x. abs x <= inv (&2 pow 2)
        ⟹ abs(cos x - poly [&1; &0; --&1 / &2; &0; &1 / &24; &0;
                            --&1 / &720; &0; &1 / &40320] x)
            <= inv(&2 pow 35)
```

For efficiency of exploration, shadow functions are also provided, which produce the polynomials as lists of numbers without performing proof, but in principle these are dispensable. In order to provably bound the accuracy of a polynomial approximation to any mathematical function, the only work required of the user is to provide these functions. For most of the common transcendental functions this is, as here, straightforward. Exceptions are the tangent and cotangent, for which arriving at the power series is a considerable amount of work.

In order to arrive at a bound on the gap between the mathematical function $f(x)$ and the polynomial approximation $p(x)$, the HOL bounding rule uses the Taylor series function to approximate $f(x)$ by a truncated Taylor series $t(x)$. If the bound is required to accuracy ϵ, the Taylor series is constructed so $|f(x) - t(x)| \leq \epsilon/2$ over the interval. Then, the remaining problem is to bound $|t(x) -$

$p(x)|$ to the same accuracy $\epsilon/2$. Since $t(x)-p(x)$ is just a polynomial with rational coefficients, this part can be automated in a regular way. In fact, the polynomial-bounding routine can be used separately, and is used at another point in this proof. The approach used is a little different from that described in [8], though the way it is used in the proof is the same.

The fundamental fact underlying the polynomial bounding rule is that the maximum of a polynomial (as for any differentiable function) lies either at one of the endpoints of the interval or at a point of zero derivative. This is proved in the following HOL theorem, which states that if a function f differentiable for $a \leq x \leq b$ has the property that $f(x) \leq K$ at all points of zero derivative, as well as at $x = a$ and $x = b$, then $f(x) \leq K$ everywhere.

```
|- (∀x. a <= x ∧ x <= b ⟹ (f diffl (f' x)) x) ∧
   f(a) <= K ∧ f(b) <= K ∧
   (∀x. a <= x ∧ x <= b ∧ (f'(x) = &0) ⟹ f(x) <= K)
   ⟹ (∀x. a <= x ∧ x <= b ⟹ f(x) <= K)
```

This reduces our problem to finding the points of zero derivative, where the derivative is another polynomial with rational coefficients. (Of course, in practice we can only isolate the roots of the polynomial to arbitrary accuracy, rather than represent them with rational numbers. However as we shall see later it is easy to accommodate the imperfect knowledge.) This is done via a recursive scheme, based on another fundamental fact: for any differentiable function f, $f(x)$ can be zero only at one point between zeros of the derivative $f'(x)$. More precisely, if $f'(x) \neq 0$ for $a < x < b$ then if $f(a)f(b) \geq 0$ there are no points of $a < x < b$ with $f(x) = 0$:

```
|- (∀x. a <= x ∧ x <= b ⟹ (f diffl f'(x))(x)) ∧
   (∀x. a < x ∧ x < b ⟹ ¬(f'(x) = &0)) ∧
   f(a) * f(b) >= &0
   ⟹ ∀x. a < x ∧ x < b ⟹ ¬(f(x) = &0)
```

and on the other hand if $f(c)f(d) \leq 0$ for any $a \leq c \leq d \leq b$ then any point $a < x < b$ with $f(x) = 0$ must in fact have $c \leq x \leq d$:

```
|- (∀x. a <= x ∧ x <= b ⟹ (f diffl f'(x))(x)) ∧
   (∀x. a < x ∧ x < b ⟹ ¬(f'(x) = &0))
   ⟹ ∀c d. a <= c ∧ c <= d ∧ d <= b ∧
            f(c) * f(d) <= &0
            ⟹ ∀x. a < x ∧ x < b ∧ (f(x) = &0)
                   ⟹ c <= x ∧ x <= d
```

Using this theorem, it is quite easy to isolate all points x_i with $f(x_i) = 0$ within arbitrarily small intervals with rational endpoints, and prove that this does indeed isolate all such points. We simply do so recursively for the derivative f' (since root isolation is trivial for constant or even linear polynomials, the

recursion is well-founded). Conservatively, we can include all isolating intervals for f''s zeros in f's, to avoid the difficulty of analyzing inside these intervals; later these can be pruned. Now if two adjacent points with $f'(x_i) = 0$ and $f'(x_{i+1})$ have been isolated by $c_i \leq x_i \leq d_i$ and $c_{i+1} \leq x_{i+1} \leq d_{i+1}$, we need only consider — assuming $d_i < c_{i+1}$ since otherwise there is no question of a root between them — whether $f(d_i)f(c_{i+1}) \leq 0$. Since by the inductive hypothesis, there are no roots of f' inside the interval $d_i < x < c_{i+1}$, we can use the above theorems to find that there are either no roots or exactly one. If there is one, then (without proof) we can isolate it within a subinterval $[c, d]$ by binary chop and then use the second theorem above to show that this isolates all roots with $d_i < x < c_{i+1}$.

Although this procedure is conservative, that doesn't matter for the use we will make of it — proving that all the roots have been isolated, even if we overcount, still makes the procedure of bounding the function over these isolating intervals a sound approach. Nevertheless, we do prune down the initial set when it is clear that the function could not cross zero within the interval. This follows naturally from interlocking recursions down the chain of derivatives f, f', f'', $\ldots$, $f^{(n)}$ evaluating both bounds and isolating intervals for all zeros. The key theorem here is:

```
|- (∀x. a <= x ∧ x <= b ⟹ (f diffl (f' x)) x) ∧
   (∀x. a <= x ∧ x <= b ⟹ (f' diffl (f'' x)) x) ∧
   (∀x. a <= x ∧ x <= b ⟹ abs(f''(x)) <= K) ∧
   a <= c ∧ c <= x ∧ x <= d ∧ d <= b ∧ (f'(x) = &0)
   ⟹ abs(f(x)) <= abs(f(d)) + (K / &2) * (d - c) pow 2
```

which allows us to go from a bound on f'' and root isolation of f' to a bound on f (a sharp second-order one using the fact that the f' has a zero in the interval to show that f is flat there). As shown above, we can easily pass from root isolation of f' to root isolation of f. Hence we have a recursive procedure to bound f by recursively bounding and isolating f' and all additional derivatives. This is all done entirely automatically and HOL constructs a proof.

5.2 Bounding the rounding error

Most of the rounding errors in the polynomial computation can be bounded simply using the '$(1+\epsilon)$' theorem or its absolute error analog. This is completely routine, and in fact has been automated [10]. However, there is one point in the sine and cosine calculations where a trick is used, and manual intervention in the proof is required. (However, for many other functions which don't use tricks, this part of the proofs *is* essentially automatic.) Without special measures, the rounding error in computing the second term of the series, PP_1r^3 in the case of sine ($PP_1 \approx \frac{1}{3}$, or QQ_1r^2 in the case of cosine ($QQ_1 = \frac{1}{2}$), would reduce the quality of the answer beyond acceptable levels. Therefore, these are computed in a more sophisticated way.

In order to reduce rounding error, r is split, using explicit bit-level operations, into two parts $r_{hi} + r_{lo}$, where r_{hi} has only 10 significant bits and r_{lo} is the corresponding "tail". For cosine, the required $\frac{1}{2}r^2$ can be computed by:

$$\begin{aligned}\frac{1}{2}r^2 &= \frac{1}{2}r_{hi}^2 + \frac{1}{2}(r^2 - r_{hi}^2) \\ &= \frac{1}{2}r_{hi}^2 + \frac{1}{2}r_{lo}(r + r_h i)\end{aligned}$$

Because r_{hi} only has 10 significant digits, its squaring commits no rounding error, and of course neither does the multiplication by a power of 2. Thus, the rounding error is confined to the much smaller quantity $r_{lo}(r+r_h i)$, which is well within acceptable limits. For sine a similar trick is used, but since the coefficient is no longer a power of 2, it is also split. We will not show the details here.

6 Final correctness theorem

The final general correctness theorems we derive have the following form:

```
|- x ∈ floats Extended ∧ abs(Val x) <= &2 pow 64
   ⟹ prac (Extended,rc,fz) (fcos rc fz x) (cos(Val x))
            (#0.07341 * ulp(rformat Extended) (cos(Val x)))
```

The function `prac` means 'pre-rounding accuracy'. The theorem states that provided x is a floating point number in the double-extended format, with $|x| \leq 2^{64}$ (a range somewhat wider than needed), the result excluding the final rounding is at most 0.07341 units in the last place from the true answer of $cos(x)$. This theorem is generic over all rounding modes `rc` and flush-to-zero settings `fz`. An easy corollary of this is that in round-to-nearest mode without flush-to-zero set the maximum error is 0.57341 ulps, since rounding to nearest can contribute at most 0.5 ulps. In other rounding modes, a more careful analysis is required, paying careful attention to the formal definition of a 'unit in the last place'. The problem is that the true answer and the computed answer before the final rounding may in general lie on opposite sides of a (negative, since $|cos(x)| \leq 1$) power of 2. At this point, the gap between adjacent floating point numbers is different depending on whether one is considering the exact or computed result. In the case of round-to-nearest, however, this does not matter since the result will always round to the straddled power of 2, bringing it even closer to the exact answer.

7 Conclusions

We have presented a representative example of the work involved in verifications of this kind. As can be seen, the mathematical apparatus necessary for the verifications is quite extensive, and we require both abstract pure mathematics

and very concrete results about floating point rounding. Moreover, since some parts of the proof, such as bounding approximation errors and routine rounding errors, would be extremely tedious by hand, programmability of the underlying theorem prover is vital. While these proofs are definitely non-trivial, modern theorem provers such as HOL Light have reached a stage of development (particularly in the formalization of the underlying mathematical theories) where they are quite feasible.

References

1. D. Bailey, P. Borwein, and S. Plouffe. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation*, 66:903–913, 1997.
2. A. Baker. *A Consise Introduction to the Theory of Numbers*. Cambridge University Press, 1985.
3. G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.
4. T. J. Dekker. A floating-point technique for extending the available precision. *Numerical Mathematics*, 18:224–242, 1971.
5. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
6. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 2nd edition, 1994.
7. J. Harrison. HOL Light: A tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
8. J. Harrison. Verifying the accuracy of polynomial approximations in HOL. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 137–152, Murray Hill, NJ, 1997. Springer-Verlag.
9. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998. Revised version of author's PhD thesis.
10. J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, 1999. Springer-Verlag.
11. J. Harrison, T. Kubaska, S. Story, and P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, 1999-Q4:1–7, 1999. This paper is available on the Web as `http://developer.intel.com/technology/itj/q41999/articles/art_5.htm`.
12. O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.
13. M. E. Remes. Sur le calcul effectif des polynomes d'approximation de Tchebichef. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 199:337–340, 1934.
14. P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, 1974.

15. S. Story and P. T. P. Tang. New algorithms for improved transcendental functions on IA-64. In I. Koren and P. Kornerup, editors, *Proceedings, 14th IEEE symposium on on computer arithmetic*, pages 4–11, Adelaide, Australia, 1999. IEEE Computer Society.
16. P. T. P. Tang. Table-lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the* 10^{th} *Symposium on Computer Arithemtic*, pages 232–236, 1991.
17. P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993. See also the CAML Web page: `http://pauillac.inria.fr/caml/`.