Isolating critical cases for reciprocals using integer factorization

John Harrison
Intel Corporation, JF1-13
2111 NE 25th Avenue
Hillsboro OR, USA
johnh@ichips.intel.com

Abstract

One approach to testing and/or proving correctness of a floating-point algorithm computing a function f is based on finding input floating-point numbers a such that the exact result $f(a)$ is very close to a "rounding boundary", i.e. a floating-point number or a midpoint between them. In the present paper we show how to do this for the reciprocal function by utilizing prime factorizations. We present the method and show examples, as well as making a fairly detailed study of its expected and worst-case behavior. We point out how this analysis of reciprocals can be useful in analyzing certain reciprocal algorithms, and also show how the approach can be trivially adapted to the reciprocal square root function.

1 Background

Suppose we have a floating-point algorithm computing a function that approximates a true mathematical function $f : \mathbb{R} \to \mathbb{R}$. For example, consider the following algorithm for the Intel® Itanium® architecture designed to compute a floating-point square root $\sqrt{a}$ using an initial reciprocal square root approximation followed by a sequence of fused multiply-adds. (In the actual implementation, the initial approximation instruction deals with special cases including $a = 0$.)

```
1.  $y_0 = \texttt{frsqrta}(a)$
2.  $H_0 = \frac{1}{2}y_0$                $S_0 = ay_0$
3.  $d_0 = \frac{1}{2} - S_0H_0$
4.  $H_1 = H_0 + d_0H_0$         $S_1 = S_0 + d_0S_0$
5.  $d_1 = \frac{1}{2} - S_1H_1$
6.  $H_2 = H_1 + d_1H_1$         $S_2 = S_1 + d_1S_1$
7.  $d_2 = \frac{1}{2} - S_2H_2$         $e_2 = a - S_2S_2$
8.  $H_3 = H_2 + d_2H_2$         $S_3 = S_2 + e_2H_2$
9.  $e_3 = a - S_3S_3$
10. $S = S_3 + e_3H_3$
```

If an algorithm is, like this one, implemented by composing basic floating-point operations (rather than, say, some more complicated analysis of bit-patterns), then the value computed can usually be represented as the result of rounding some approximation $f^*(x) \approx f(x)$, the value before the final rounding. In this case, the final S results from rounding the exact value $S_3 + e_3H_3$.

The algorithm will therefore round correctly for all inputs x such that $f^*(x)$ and $f(x)$ round to the same number (for all the rounding modes under consideration). In the concrete square root example, this means that $\sqrt{a}$ and $S_3 + e_3H_3$ should always round the same way.

A sufficient condition for equivalent rounding behavior is that the two values $f^*(x)$ and $f(x)$ should never be separated by a rounding boundary, i.e. a floating-point number (for directed rounding) or a midpoint (for round-to-nearest). That is, there is never a rounding boundary m with $f(x) \leq m \leq f^*(x)$ or $f^*(x) \leq m \leq f(x)$, unless $f^*(x) = f(x)$. (Not quite a necessary condition in the round-to-nearest mode since if one is exactly equal to the rounding boundary and the other on the "right" side, the correct result will be obtained.) This is usually hard to establish by analytic reasoning. However, it is usually easy to establish some sort of relative error bound ϵ such that:

$$|f^*(x) - f(x)| \leq \epsilon|f(x)|$$

Therefore, misrounding can occur only when

$$|f(x) - m| \leq \epsilon|f(x)|$$

It is therefore interesting for purposes of both testing and proving correctness to deliberately concoct test points x to make the relative distance from a rounding boundary $|f(x) - m|/|f(x)|$ as small as possible. Indeed, irrespective of the details of the algorithms we are concerned with, these test points might be expected to display greatest sensitivity to the accuracy of $f^*(x)$ and so show up errors most easily.

For some basic algebraic functions, such special x can be found analytically using number-theoretic techniques [14, 11], in such a way that the very worst examples (having the smallest relative distance from a rounding boundary) are isolated. For transcendental functions, this is more difficult, but one can still generate good cases by exploiting local linearity and solving congruences. For double-precision it is feasible, though costly, to isolate the very worst examples [6].

One use of the points so obtained is to test floating-point functions. Indeed, Parks [11] reports that such testing exposed a bug in a commercial microprocessor. A more ambitious goal, realized for square root algorithms by Cornea [1], is to isolate a sufficiently large set of points that the correct behavior of the algorithm on these, in conjunction with an analytical proof that covers all other cases, gives a complete correctness proof of the algorithm in all cases. For example, if we can prove analytically that for all floating-point numbers x we have:

$$|f^*(x) - f(x)| \leq \epsilon |f(x)|$$

and that some set S_ϵ contains all points x where $|m - f(x)| \leq \epsilon |f(x)|$ for some rounding boundary m, the correctness of the algorithm in all cases is equivalent to the correctness just for the points in S_ϵ. If such sets can be found easily and they are not too large, this gives a very effective methodology for proofs of algorithms. The goal of this paper is to show how to isolate such special cases for the reciprocal (and reciprocal square root) function and demonstrate their applicability in such correctness proofs of algorithms.

2 Critical cases for quotient and reciprocal

We will in what follows consider a single floating-point format with precision p, which contains all the floating-point numbers concerned and is also the destination format for the result. We also ignore the possibility of overflow and underflow in computation sequences. This keeps the presentation simpler and accords well with the intended applications where all input numbers are double-extended and additional exponent range (but not precision) is available for intermediate computations. The results that follow can straightforwardly be refined for mixed-precision applications.

It's instructive to examine the problem for the general case of quotients, and then contrast the restriction to the reciprocal. In general, we seek floating-point numbers x and y such that x/y lies close to some w that is either itself a floating-point number or a midpoint between two floating-point numbers. Without loss of generality, we can assume:

$$\begin{array}{ll} x = 2^{e_x} a & 2^{p-1} \leq a < 2^p \\ y = 2^{e_y} b & 2^{p-1} \leq b < 2^p \\ w = 2^{e_w} m & 2^p \leq m < 2^{p+1} \end{array}$$

where p is the floating-point precision and a, b and m, as well as the various e_i, are integers. Note that even values of m correspond to floating-point numbers and odd values correspond to midpoints. We are interested in how small the relative difference $|x/y - w|/|x/y|$ can become. This relative difference can be rewritten as:

$$\frac{|x/y - w|}{|x/y|} = |1 - wy/x| = |1 - 2^{-q} mb/a|$$

where $q = e_x - (e_w + e_y)$, and so

$$\frac{|mb - 2^q a|}{2^q a}$$

Given the ranges of the values a, b and m, we have

$$2^{2p-1} \leq mb < 2^{2p+1}$$

and

$$2^{q+p-1} \leq 2^q a < 2^{q+p}$$

It turns out that the only interesting cases are when $q = p$ or $q = p + 1$. For if $q \leq p - 1$ then $q + p \leq 2p - 1$ so we have

$$2^q a \leq 2^q (2^p - 1) < 2^{2p-1} \leq mb$$

(remember that the values a, b and m are integers so when $< 2^r$ they are actually $\leq 2^r - 1$) and so

$$\frac{|mb - 2^q a|}{2^q a} \geq 2^q/(2^q a) = 1/a > 2^{-p}$$

.
Similarly if $q = p + 2$ we have:

$$mb \leq (2^p - 1)(2^{p+1} - 1) < 2^{2p+1} \leq 2^{p+2} a < 2^{2p+2}$$

and therefore

$$\frac{|mb - 2^q a|}{2^q a} \geq (2^{p+1} + 2^p - 1)/(2^q a) > 2^{p+1}/2^{2p+2} = 2^{-(p+1)}$$

Finally, if $q \geq p + 3$ then $2^q a > 2mb$ and so

$$\frac{|mb - 2^q a|}{2^q a} > 1/2$$

In all these cases, the distance is at least $2^{-(p+1)}$. Therefore, when seeking cases where the distance is of order 2^{-2p} (for realistic p) we need only consider $q \in \{p, p+1\}$. This being the case, the denominator $2^q a$ is constrained to within a factor of 4, so the essential problem is to find how small

$$|mb - 2^q a|$$

can become for $q \in \{p, p+1\}$. Since the value is an integer, we can try to find small values by explicit consideration of the various possibilities in succession:

$$\begin{aligned}
mb &= 2^p a + 1 \\
mb &= 2^p a - 1 \\
mb &= 2^{p+1} a + 1 \\
mb &= 2^{p+1} a - 1 \\
mb &= 2^p a + 2 \\
mb &= 2^p a - 2 \\
mb &= 2^{p+1} a + 2 \\
mb &= 2^{p+1} a - 2 \\
mb &= 2^p a + 3 \\
&\ldots
\end{aligned}$$

It seems that the number of possible solutions of these equations is too large for this to be a practical approach. On the other hand, if we fix any one of the values a, b and m, the problem becomes tractable. If we fix either m or b then the problem becomes a set of linear congruences (with additional range restrictions filtering the possible solution set), which are easy to solve. If we consider the special case of the reciprocal, then we fix $a = 2^{p-1}$. This problem is also tractable, as we shall see, but has a somewhat different character. We just need to consider

$$\begin{aligned}
mb &= 2^{2p-1} + \delta \\
mb &= 2^{2p} + \delta
\end{aligned}$$

for successive small integers δ. In fact, the situation is even better, because once again no small values can arise in the former case because of the range limitation, except for the trivial $mb = 2^{2p-1}$; the next case must be $(2^p + 1)2^{p-1} = 2^{2p-1} + 2^{p-1}$. So we need only be concerned with solutions to

$$mb = 2^{2p} + \delta$$

for integers $2^{p-1} \leq b < 2^p$ and $2^p \leq m < 2^{p+1}$. Indeed, for small δ, it is easy to see that the two upper bounds imply the lower ones.

3 Factorization distribution

Our approach to the problem of finding all solutions to $mb = 2^{2p} + \delta$ (with p and δ fixed) is quite straightforward. We find the prime factorization of $2^{2p} + \delta$, and consider all possible ways of distributing these prime factors into two parts m and b subject to the appropriate range limitation $m < 2^{p+1}$ and $b < 2^p$. In general, we will refer to a factorization $n = ab$ of n with $a < A$ and $b < B$ as an (A, B)-*balanced* factorization.

Consider, for illustration, the case $p = 6$ and $\delta \in \{\pm 1, \pm 2, \pm 3\}$. In each case we find the prime factorization of $2^{2p} + \delta$:

$$\begin{aligned}
2^{12} + 1 &= 17 \cdot 241 \\
2^{12} - 1 &= 3^2 \cdot 5 \cdot 7 \cdot 13 \\
2^{12} + 2 &= 2 \cdot 3 \cdot 683 \\
2^{12} - 2 &= 2 \cdot 23 \cdot 89 \\
2^{12} + 3 &= 4099 \\
2^{12} - 3 &= 4093
\end{aligned}$$

In the cases $2^{12}+1$, $2^{12}+2$, $2^{12}+3$ and $2^{12}-3$, the largest factor is already $> 2^{p+1} = 128$, so there is no possible distribution obeying the range restrictions. For $2^{12} - 2$ there is exactly one such distribution:

$$m \cdot b = 89 \cdot (2 \cdot 23) = 89 \cdot 46$$

Note that the 'symmetrical' distribution is not admissible because $89 > 2^p$. For $2^{12} - 1$, there are four possible distributions:

$$\begin{aligned}
m \cdot b &= (3^2 \cdot 13) \cdot (5 \cdot 7) = 117 \cdot 35 \\
m \cdot b &= (3 \cdot 5 \cdot 7) \cdot (3 \cdot 13) = 105 \cdot 39 \\
m \cdot b &= (7 \cdot 13) \cdot (3^2 \cdot 5) = 91 \cdot 45 \\
m \cdot b &= (5 \cdot 13) \cdot (3^2 \cdot 7) = 65 \cdot 63
\end{aligned}$$

Note that the corresponding m are all odd, and therefore represent midpoints. Thus, we can say that $|1/y - w| \geq 4/2^{12}|1/y|$ for any midpoint w except in the cases where y's significand b is in the set $\{35, 39, 45, 46, 63\}$; for $b = 46$ we get a $2/2^{12}$ relative distance and for 35, 39, 45 and 63 we get $1/2^{12}$. Since the above lists exhausts all m, even or odd, we see that $|1/y - w| \geq 4/2^{12}|1/y|$ for any floating-point number w, except for the special cases when y is a power of 2 and so its reciprocal is exactly representable (i.e. $1/y = w$).

4 Implementation

The implementation of the above idea is straightforward, given any reasonable programming language. We have used Objective CAML, a very high-level functional language that we have previously used extensively for implementation of theorem proving code:

`http://www.ocaml.org/`

This already has a multiprecision integer and rational function datatype available. It does not, however, have a built-in library for factoring numbers, and we did not want to write our own code for this operation — since the numbers can be as large as 2^{226} (for quad precision reciprocals), factorization is a non-trivial problem. We used the factoring code included in the PARI / GP system:

`http://www.parigp-home.de/`

The documentation says:

factorint$(n, \{flag = 0\})$: factors the integer n using a combination of the Shanks SQUFOF and Pollard Rho method (with modifications due to Brent), Lenstra's ECM (with modifications by Montgomery), and MPQS (the latter adapted from the LiDIA code with the kind permission of the LiDIA maintainers), as well as a search for pure powers with exponents≤ 10.

We are not experts in the topic of factorization, but have been quite impressed with how fast it usually factors numbers. Only for quad precision, when the numbers are of the order 2^{226}, does it start to slow down noticeably. Rather than a strict primality test, the factors are only subjected to a strong probabilistic primality test. Therefore, out of paranoia, we have developed our own code to certify primality, by constructing prime certificates in the style of Pratt [12], appealing to Lucas's theorem. That is, to certify that each p occurring in PARI/GP's factorization is prime, we show that there is a primitive root a modulo p such that $a^{p-1} \equiv 1 \pmod p$ but $a^{\frac{p-1}{q}} \not\equiv 1 \pmod p$ for any prime factor q of $p-1$. (The primitive root a is found randomly, and the factors q of $p-1$ are found by using PARI/GP's factorization recursively, certifying those factors as primes too.) This certification slows down the factorization process by a moderate amount, so we sometimes switch it off when experimenting.

Once we have the prime factors, we need to test all ways of distributing them over two numbers subject to range restrictions. As noted, we need only apply the upper range restrictions $m < 2^{p+1}$ and $b < 2^p$. Roughly, we just naively enumerate all possibilities. In order to cut off choice points as soon as possible, we start distributing from the largest prime factors, i.e. consider the prime factors $p_1^{\alpha_1} \cdot p_2^{\alpha_2} \cdot \cdots \cdot p_k^{\alpha_k}$ in decreasing order $p_1 > p_2 > \cdots > p_k$. We first consider all $\alpha_1 + 1$ ways of distributing $p_1^{\alpha_1}$ into two parts. If any of these distributions already violate the range restriction, they are abandoned. Otherwise, for each one, we consider the $\alpha_2 + 1$ ways of distributing $p_2^{\alpha_2}$, and so on. The algorithm is very straightforward to program recursively in OCaml.

It might be doubted whether such a naive distribution algorithm is acceptably efficient. At least it has been adequate to obtain some results quite quickly for the main precisions that interest us, $p \in \{24, 53, 63, 113\}$. We first look at some of these results and then turn to a detailed performance analysis.

5 Results

Table 1 presents a small sample of the results obtained using the methods outlined above. For each of the four major precisions $p = 24, 53, 64, 113$, we list the 66 floating-point significands whose reciprocals are closest either to floating-point numbers or midpoints. This distance, as a multiple of the corresponding 2^{-2p}, is given in the 'd' columns. When, as often happens, several reciprocals have the same 'd' value we order them in decreasing order, and cut the table off on that basis. The asterisk means that the distance is from a floating-point number (and hence may be unimportant if we are concerned only with round-to-nearest).

Larger lists for d up to a few thousand can be generated for all these precisions without requiring more than a few days of runtime on a modern machine. And of course, it is trivial to parallelize the task since it consists of a separate subtask for each d considered.

6 Applications

We can use the techniques set out above in the design and verification of algorithms for correctly rounded reciprocals. These might be substituted by the programmer, or by the compiler if it can recognize that in an expression a/b, the constant a is guaranteed to be 1. (This could be generalized to any power of 2.) For example, the following algorithm is normally used for double-extended precision division (precision $p = 64$) on Intel® Itanium® processors.

```
1. $y_0 = \mathtt{frcpa}(b)$
2. $d = 1 - by_0$              $q_0 = ay_0$
3. $d_2 = dd$                  $d_3 = dd + d$
4. $y_1 = y_0 + y_0 d_3$       $d_5 = d_2 d_2 + d$
5. $y_2 = y_0 + y_1 d_5$       $r_0 = a - bq_0$
6. $e = 1 - by_2$              $q_1 = q_0 + r_0 y_2$
7. $y_3 = y_2 + ey_2$          $r = a - bq_1$
8. $q = q_1 + ry_3$
```

Single precision		*Double precision*		*Extended precision*		*Quad precision*	
Mantissa	*d*	Mantissa	*d*	Mantissa	*d*	Mantissa	*d*
0x800000	0*	0x10000000000000	0*	0x8000000000000000	0*	0x10000000000000000000000000000	0*
0xFFFFFF	1	0x1FFFFFFFFFFFFF	1	0xFFFFFFFFFFFFFFFF	1	0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFF	1
0xFE01FF	1	0x1FFFFFF8000001	1	0xD6329033D6329033	1	0x1FFFFFFFFFFFFFE00000000000001	1
0xFC3237	1	0x1FD8CD299E8D79	2*	0xB7938C6947D97303	1	0x1B52F1BB6F8DC3F0D920E2F3D449B	1
0xF0FF0F	1	0x1FC94266515BC9	2*	0x99D0C486A0FAD481	1	0x19C1ECF3420D27F8729BA7E1AB31D	1
0xF02A3B	1	0x1F739BD459BEA2	2	0x989E556CADAC2D7F	1	0x17ABDE305BAC595488190B4AD7657	1
0xF00FF1	1	0x1F65FAD23B0D86	2	0x8E05E117D9E786D5	1	0x14367E6C7D1CD9E2833D2900EE8D5	1
0xEE4BC5	1	0x1EF7930608393E	2	0xFFFFFFFFFFFFFFFE	4*	0x1FFBA28E4810FB56A9FDD85058227	2*
0xEC7EC7	1	0x1EDA43AEE3120F	2*	0xFFFFFFFE00000002	4*	0x1FF0231E35F73DFF14F89AADF10C2	2
0xE25473	1	0x1ED31F284BA183	2*	0xFFFFFF000007FFFFE	4*	0x1FE5A1913A4EF66DEF762D8053282	2
0xE1368B	1	0x1E9A9473949BF6	2	0xFF801FFA00FFE002	4*	0x1FE1696E4EFFB6A84655C0D432D92	2
0xE05475	1	0x1E8D517D09C5C2	2	0xFF007FC01FF007FE	4*	0x1FDA070427995BB524AB4B13DC457	2*
0xDE86A9	1	0x1E756F08DF1792	2	0xFE421D63446A3B34	4	0x1FAA42B2AE532A32F819FE18EDEAF	2*
0xDC23DD	1	0x1E4599DD926B71	2*	0xFDC1EAD583108905	4*	0x1F97117BE0A19F4B8279CEBB8A682	2
0xD43D43	1	0x1E20ADBC4078A2	2	0xFC41DF1077C41DF3	4*	0x1F76B18346B7182CE92732C773FB2	2
0xD25D25	1	0x1DE4A0D00FA9B2	2	0xFC07FFE03FFF01FE	4*	0x1F742DB89E4A0B81D5A2FE647E4EB	2*
0xD0DD0D	1	0x1DE441D5331432	2	0xFC07F01FC07F01FE	4*	0x1F490212CC8000000003E92042599	2*
0xD0AC19	1	0x1DA83EEDD80267	2*	0xFBFC17DFE0BEFF04	4	0x1F40436566B31BF75C99DF44F291F	2*
0xC23DC3	1	0x1DA210DAEB138E	2	0xFB20F95555168D17	4*	0x1F361A9D498B669732A60AFCF9461	2*
0xC100C1	1	0x1D7F8AC20F7A3F	2*	0xFB0089D7241D10FC	4	0x1F25136B121FE2DD08B9975B8DBD2	2
0xB84A93	1	0x1D7B72B82BAE23	2*	0xFA0BF7D05FBE82FC	4	0x1F1DFB37ABDE94B6800C5550CD152	2
0xAD1367	1	0x1D5B9032F086BE	2	0xF98AF433A85E62BF	4*	0x1F182E16A52503DCEDEBC24CA2B4E	2
0xAB8BE1	1	0x1D5616F7BA44B9	2*	0xF96BA24DC930852A	4*	0x1F140333F6B5946A06272DBD508B7	2*
0xA6449F	1	0x1C8ECFA282734B	2*	0xF93AB02081C9D1D6	4*	0x1F0DD51725F05CC5C752AA05A4311	2*
0xA24CF7	1	0x1C69BF28EBA166	2	0xF912590F016D6D04	4	0x1F001BEA0DE009CE597A0A8CE4B02	2
0xA0DDD1	1	0x1C67CF42F20D11	2*	0xF858A9FE5A20550D	4*	0x1EC36516E1240A243EF66232D4BA7	2*
0x9BEAAF	1	0x1C4D3AABD478F6	2	0xF84CEE8E701FC266	4*	0x1EBC4C4507F9CE8304761C8F703D2	2
0x986799	1	0x1C2693DCF34742	2	0xF774DD7F912E1F54	4	0x1E9EDBD047D1D813FB315AB469B2E	2
0x909909	1	0x1BEA3278B789D2	2	0xF7444DFBF7B20EAC	4	0x1E89A9332E4A8C84E2AA22A6DF7F1	2*
0x8EFA43	1	0x1BB2278C9B2F97	2*	0xF6F0243D8121FB7A	4*	0x1E8119576512C73436A03607DCB9B	2*
0x87CC45	1	0x1B962F9EBB9659	2*	0xF6640F754B4E709A	4*	0x1E765D90D920CEEEBD7F5E0E0BBA9	2*
0x869913	1	0x1B227794E85702	2	0xF39EB657E24734AC	4	0x1E5DF4E7C4BB29C00588956CF0009	2*
0xCA6691	2*	0x1B0FD7099EB189	2*	0xF36EE790DE069D54	4	0x1E587973506590E472C4A72A35CF1	2*
0xA1E58F	2*	0x1B0942AAAE0BD3	2*	0xF363A464E2DCD8EB	4*	0x1E40DECFCF36257C367CACDAD3F77	2*
0xFFFFFE	4*	0x1AE6849E786AD2	2	0xF286AD7943D79434	4	0x1E2BE9D384CE2D85FD8013E21ECF2	2
0xFFE002	4*	0x1ABBEB8E009CE1	2*	0xEF9DA1D868469215	4*	0x1E15BB4DD7AA987E15487C533C649	2*
0xAAAAAC	4	0x1AA7C88EE59082	2	0xEDF09CCC53942014	4	0x1E0C9181ECD8418355A1A49887852	2
0x8C1284	4	0x1A6F41DAB98CB2	2	0xEDE957FFFFC485AA	4*	0x1E0697C8651B43A9309DE9E6F021E	2
0x801001	4*	0x1A2CE4D7478A06	2	0xEDE95090B57B7A56	4*	0x1DEE59C5D9CC8CB8613C8C6AD453F	2*
0x800001	4*	0x1A149BAD85DE72	2	0xEDBAA0922AFB6EAA	4*	0x1DBA39CE33CA8DEF599F5DA2A534F	2*
0x94F105	5	0x1A0E795098FF63	2*	0xEC4B058D0F7155BC	4	0x1DAC85098ABA5E144E44187FB5467	2*
0x92ABAB	5	0x1A0B8FFFFCBE8E	2	0xEC1CA6DB6D7BD444	4	0x1D99C15392893EA6B5200AB3E8819	2*
0xE401C8	8	0x19F142D24E1352	2	0xEB443F5A21FAD10E	4*	0x1D983DBB99EAC626064F81D7BF4D2	2
0xE071F9	8*	0x19BD2D9FD24AD7	2*	0xEA6EE2D972746ED1	4*	0x1D86CC4938A03D4525C152AB8505E	2
0xD6D764	8*	0x19B8D7C084EE43	2*	0xEA40E197842DA6AF	4*	0x1D573D7B5CFE2D277AD5E05BAC65E	2
0xD443F2	8*	0x199E1B447E99C2	2	0xE934A8E070ACB65D	4*	0x1D4D79E1F152354E10F583D4A65C9	2*
0xCFBA38	8	0x19939800033273	2*	0xE84BDA12F684BDA3	4*	0x1D4CF86C34F75247D8FA16202FA29	2*
0xB5C2F1	8*	0x1975E059B82E49	2*	0xE775FF856986AE74	4	0x1D4562A76F879A38EEE86F526D231	2*
0xB447BC	8*	0x1960A45D1A71E6	2	0xE6944AE6502F8A22	4*	0x1D3168C71EC1068F69A433D0DF9B7	2*
0xAE4F88	8	0x19385F4F83B2B1	2*	0xE5CB972E5CB972E4	4	0x1D26EDCA8F70B604EC3E7797A93A1	2*
0x9A5F6E	8*	0x190759A7F39561	2*	0xE597116BD81B26A3	4*	0x1D16B51A196CFFF2477078355A9AE	2
0x988597	8*	0x18F187FFFCE1CF	2*	0xE58C38D1342FBE3A	4*	0x1D12CF093CC27703DA21DC7D68CE7	2*
0x91FEDC	8*	0x18DFA37A569E47	2*	0xE58469F0234F72C4	4	0x1CEC5ADBF01E9685CD487AD8F3327	2*
0xFFFFFD	9	0x1879574AF5FBB1	2*	0xE511C4648E2332C4	4	0x1CE72221273FE0035FEC64CBB3DBF	2*
0xA013D1	11	0x184A12EFEF626E	2	0xE3FC771FE3B8FF1C	4	0x1CE4C2D686D170738B75E2AFECF3E	2
0xF56DA7	12*	0x18401CBCDB5596	2	0xE3C845B18BD25EC6	4*	0x1CD0C2468D84F6ACF871D5E1FCBA9	2*
0x858376	12*	0x181EFE51EAD722	2	0xE318DE3C8E6370E4	4	0x1CCB50FE42CD1B95A59CA8AD6EB99	2*
0xCF7D05	13	0x1806C89FCB9452	2	0xE301201062C997DE	4*	0x1CABCCB01B54CE7E2A63A99B9D9C2	2
0xEE5223	14*	0x17F52093014F0E	2	0xE23B9711DCB88EE4	4	0x1C9ED60CFD93F55F117571C3FDA0E	2
0xE528AB	14*	0x17D93736C115FF	2*	0xE231188C46231187	4*	0x1C820E19C1A66CE04C62A562E9111	2*
0xBF3621	14*	0x17A6B0778D60C1	2*	0xE1F00785C1FF0F82	4*	0x1C673D52FCD6E005D2A2B3D40EDCF	2*
0xAB5ED9	14*	0x178CB7D5D6E322	2	0xE159BE4A8763011C	4	0x1C670773DF1678DF0A4336D3FE21E	2
0x8EFE15	14*	0x17641C46F799EE	2	0xE0A72F05397829CA	4*	0x1C4D4290F01337DEE39B3A7862BDE	2
0x897ECD	14*	0x175D929C3C2FC9	2*	0xE0A720FAC6F829CA	4*	0x1C4BD3136A6DB6DB6DB351F3546E2	2
0xFFFFFC	16*	0x1733B8284238F1	2*	0xE073C0EE938231F9	4*	0x1C49777FF62E0B0DA1A4CDB58587F	2*
0xF83F04	16*	0x17255CA25B68E1	2*	0xDF738B7CF7F482E4	4	0x1C4814DFE06ECB4EB88D00BA934F2	2

Table 1. Some numbers with reciprocals closest to numbers(*) and midpoints

As usual in algorithms of this kind, each operation uses a fused multiply-add (*not* a separate multiplication and addition), all steps but the last are performed in round-to-nearest mode with additional exponent range precluding the possibility of intermediate overflow or underflow, and the last operation is done in the intended rounding mode and target precision.

Embedded in this algorithm is the computation of a very accurate reciprocal approximation y_3. Originally, in the design of algorithms of this kind, the correctness of the final rounding of q was justified by a theorem whose precondition requires perfect rounding of y_3 [9], and only later was it noted by the present author that a relative error $y_3 = \frac{1}{b}(1+\epsilon)$ for $|\epsilon| < 2^{-p}$ suffices, which can be satisfied by a relatively weak error condition on y_2 and the analysis of a few special cases [3, 8]. However, if we are in a situation where $a = 1$ we might consider, instead of using the entire sequence, unpicking the algorithm for reciprocation to be used directly, since its latency is shorter by 1 operation, and it uses only 9 floating-point operations instead of 14:

```
1. $y_0 = \texttt{frcpa}(b)$
2. $d = 1 - by_0$
3. $d_2 = dd$                  $d_3 = dd + d$
4. $y_1 = y_0 + y_0 d_3$       $d_5 = d_2 d_2 + d$
5. $y_2 = y_0 + y_1 d_5$
6. $e = 1 - by_2$
7. $y = y_2 + ey_2$
```

Now the question of whether y is always correctly rounded becomes critical. First we will consider round-to-nearest. The initial approximation returned by `frcpa` will satisfy $y_0 = \frac{1}{b}(1+\epsilon_0)$ for some $|\epsilon_0| \leq 2^{-8.886}$. A routine relative error analysis, assuming each rounding $rn(x)$ yields $x(1+\epsilon)$ for some $|\epsilon| \leq 2^{-64}$, shows that y^*, the value of y before the last rounding, satisfies

$$y^* = \frac{1}{b}(1+\epsilon)$$

where $|\epsilon| \leq 2^{-123.37}$. Therefore, the only cases where incorrect rounding can occur are those closer than this relative distance to a midpoint. The potentially failing significands b can be isolated by finding all $(2^{65}, 2^{64})$-balanced factorizations $mb = 2^{128} + d$ for integers $|d| \leq 24$ (since $24 + 1 > 2^{-123.37}/2^{-128}$) and m odd. The set of b values that we need to consider are the following 134 (ordered in decreasing size, not according to their closeness to a midpoint):

```
0xFFFFFFFFFFFFFFFF 0xFFFFFFFFFFFFFFFD 0xFE421D63446A3B34
0xFBFC17DFE0BEFF04 0xFB940B119826E598 0xFB0089D7241D10FC
0xFA0BF7D05FBE82FC 0xF912590F016D6D04 0xF774DD7F912E1F54
0xF7444DFBF7B20EAC 0xF39EB657E24734AC 0xF36EE790DE069D54
0xF286AD7943D79434 0xEDF09CCC53942014 0xEC4B058D0F7155BC
0xEC1CA6DB6D7BD444 0xE775FF856986AE74 0xE5CB972E5CB972E4
0xE58469F0234F72C4 0xE511C4648E2332C4 0xE3FC771FE3B8FF1C
0xE318DE3C8E6370E4 0xE23B9711DCB88EE4 0xE159BE4A8763011C
0xDF738B7CF7F482E4 0xDEE256F712B7B894 0xDEE24908EDB7B894
0xDE86505A77F81B25 0xDE03D5F96C8A976C 0xDDFF059997C451E5
0xDB73060F0C3B6170 0xDB6DB6DB6DB6DB6C 0xDB6DA92492B6DB6C
0xDA92B6A4ADA92B6C 0xD9986492DD18DB7C 0xD72F32D1C0CC4094
0xD6329033D6329033 0xD5A004AE261AB3DC 0xD4D43A30F2645D7C
0xD33131D2408C6084 0xD23F53B88EADABB4 0xCCCE6669999CCCD0
0xCCCE666666633330 0xCCCCCCCCCCCCCCD0 0xCBC489A1DBB2F124
0xCB21076817350724 0xCAF92AC7A6F19EDC 0xC9A8364D41B26A0C
0xC687D6343EB1A1F4 0xC54EDD8E76EC6764 0xC4EC4EC362762764
0xC3FCF61FE7B0FF3C 0xC3FCE9E018B0FF3C 0xC344F8A627C53D74
0xC27B1613D8B09EC4 0xC27B09EC27B09EC4 0xC07756F170EAFBEC
0xBDF3CD1B9E68E8D4 0xBD5EAF57ABD5EAF4 0xBCA1AF286BCA1AF4
0xB9B501C68DD6D90C 0xB880B72F050B57FC 0xB85C824924643204
0xB7C8928A28749804 0xB7A481C71C43DDFC 0xB7938C6947D97303
0xB38A7755BB835F24 0xB152958A94AC54A4 0xAFF5757FABABFD5C
0xAF4D99ADFEFCAAFC 0xAF2B32F270835F04 0xAE235074CF5BAE64
0xAE0866F90799F954 0xADCC548E46756E64 0xAD5AB56AD5AB56AC
0xAD5AAA952AAB56AC 0xAB55AAD56AB55AAC 0xAAAAB55555AAAAAC
0xAAAAAAAAAAAAAAAC 0xAAAAA00000555554 0xA93CFF3E629F347D
0xA80555402AAA0154 0xA8054ABFD5AA0154 0xA7F94913CA4893D4
0xA62E84F95819C3BC 0xA5889F09A0152C44 0xA4E75446CA6A1A44
0xA442B4F8DCDEF5BC 0xA27E096B503396EE 0x9E9B8FFFFFD8591C
0x9E9B8B0B23A7A6E4 0x9E7C6B0C1CA79F1C 0x9DFC78A4EEEE4DCB
0x9C15954988E121AB 0x9A585968B4F4D2C4 0x99D0C486A0FAD481
0x99B831EEE01FB16C 0x990C8B8926172254 0x990825E0CD75297C
0x989E556CADAC2D7F 0x97DAD92107E19484 0x9756156041DBBA94
0x95C4C0A72F501BDC 0x94E1AE991B4B4EB4 0x949DE0B0664FD224
0x942755353AA9A094 0x9349AE0703CB65B4 0x92B6A4ADA92B6A4C
0x9101187A01C04E4C 0x907056B6E018E1B4 0x8F808E79E77A99C4
0x8F64655555317C3C 0x8E988B8B3BA3A624 0x8E05E117D9E786D5
0x8BEB067D130382A4 0x8B679E2B7FB0532C 0x887C8B2B1F1081C4
0x8858CCDCA9E0F6C4 0x881BB1CAB40AE884 0x87715550DCDE29E4
0x875BDE4FE977C1EC 0x86F71861FDF38714 0x85DBEE9FB93EA864
0x8542A9A4D2ABD5EC 0x8542A150A8542A14 0x84BDA12F684BDA14
0x83AB6A090756D410 0x83AB6A06F8A92BF0 0x83A7B5D13DAE81B4
0x8365F2672F9341B4 0x8331C0CFE9341614 0x82A5F5692FAB4154
0x8140A05028140A04 0x8042251A9D6EF7FC
```

One can show by explicit computation that the algorithm works correctly on these values. It therefore rounds correctly on all values in round-to-nearest.

For directed rounding modes, the situation is less good. Once again the relative error condition gives rise to a set of test points, this time 227 of them. The algorithm works correctly on 220 of them, but not on floating-point numbers with one of the following 7 significands, the last of these representing exact powers of 2, for which the true result is exactly representable. Cognoscenti who perform a back-of-envelope calculation will not be surprised by the failure on exactly representable results, since correctness here would require y_2 already to be the correct result, which our relative error cannot quite guarantee.

```
0x8c82da588adc6416 0x84fdf027ef813f7b 0x827b9b8059090ab2
0x8080402010080401 0x8000080000400001 0x8000000000000001
0x8000000000000000
```

This analysis indicates that the algorithm will produce correctly rounded results if the ambient rounding mode is known to be round-to-nearest, but will not always guarantee correct rounding in other rounding modes. Moreover, note that for the same reason, the 'inexact' flag will be incorrectly set in round-to-nearest mode in the special cases where b is a power of 2. (As noted, the penultimate approximation y_2 cannot be the exact reciprocal in such cases, for otherwise we would obtain $e = 0$ and correct rounding in all modes.) However, if this is considered important, it would be easy to detect and fix the problem with special case code without affecting overall latency.

7 Feasibility study

Although the previous sections show that the method is usefully applicable to some real problems, it's worth analyzing how practical the approach is likely to be in general. In attempting to use the method, three potential practical problems might arise

- Too many special points are isolated for further analysis to be feasible
- The factorization of some of the numbers is not feasible
- The distribution of prime factors is not feasible

We will not analyze the feasibility of factorization, since we do not understand the details of its implementation. We will however make the empirical observation that all factorizations for precisions up to $p = 64$ seem to be very straightforward for PARI / GP, taking a fraction of a second, while those for $p = 113$ usually take several seconds and, exceptionally, minutes.

Average density of balanced factorizations

It is not difficult to see that "on average" we obtain a fairly modest number of balanced factorizations per value examined. First note that the number of (A, B)-balanced products of numbers $\leq n$ is the number of lattice points contained both within the rectangle $0 \leq x \leq A$, $0 \leq y \leq B$ and under the curve $xy = n$. We can get a good estimate by ignoring "edge effects" and just considering the plane area, integrating to obtain:

$$C(n) = n(1 + \ln(\frac{AB}{n}))$$

Differentiating with respect to n yields the expected density, i.e. the average number of (A, B)-balanced product representations of a number close to n:

$$D(n) = \ln(AB/n)$$

Of course, these gross averages do not reflect small-scale fluctuations. Nevertheless, the agreement is fairly good with some empirical results obtained by sampling. In the following table, we examine the density of $(2^p, 2^p)$-balanced products for several p, looking in each case at 31 regions close to $\frac{k+1/2}{32} 2^{2p}$ for $0 \leq k \leq 31$ and sampling 1024 successive points in each. The final figures at the bottom give the mean value. This indicates how accurate the sampling process is on average; perfectly representative sampling would give exactly 1 here. (We avoid sampling at $\frac{k}{32} 2^p$ because that would lead to strong correlations between the sets of numbers at different k.)

$\ln(2^{2p}/n)$	$p = 24$	$p = 53$	$p = 64$
4.1588	4.4785	4.6835	3.3300
3.0602	2.8496	5.6621	3.2734
2.5494	2.4570	2.7070	2.2753
2.2129	2.0332	2.2421	2.2089
1.9616	2.0000	1.6953	2.3417
1.7609	1.9101	1.5664	1.5585
1.5939	1.5742	1.9140	1.2128
1.4508	1.3632	1.4765	1.5625
1.3256	1.3144	1.0839	1.2558
1.2144	1.2050	1.2187	1.2890
1.1143	1.0175	1.0996	1.4296
1.0233	1.0273	0.9335	0.9687
0.9400	0.7539	0.9062	0.8828
0.8630	0.7636	0.8613	0.8789
0.7915	0.6875	0.7187	0.6875
0.7248	0.6933	0.6621	0.7832
0.6623	0.6621	0.5976	0.7656
0.6035	0.5878	0.5468	0.6445
0.5479	0.5546	0.6210	0.5683
0.4953	0.4941	0.5136	0.6289
0.4453	0.4394	0.3847	0.3652
0.3976	0.3984	0.4453	0.4277
0.3522	0.3417	0.3476	0.3242
0.3087	0.3203	0.2890	0.3593
0.2670	0.2382	0.2285	0.2773
0.2270	0.2480	0.2070	0.3007
0.1885	0.1347	0.2207	0.2148
0.1515	0.1347	0.1640	0.1562
0.1158	0.0839	0.0976	0.1015
0.0813	0.0917	0.1054	0.0761
0.0480	0.0449	0.0371	0.0527
0.0157	0.0078	0.0078	0.0156
1.0000	0.9660	1.0701	0.9755

So much for the average case. What about the worst case? This seems a more difficult question to address theoretically, but in the next section we will show how to obtain a pessimistic upper bound.

Feasibility of distribution algorithm

Although the final number of values produced depends on the number of balanced factorizations, the process by which the balanced factorizations are enumerated involves examination of many dead-end paths, so the runtime of the distribution process may be very large relative to the final number of possibilities produced. A reasonable, though pessimistic, bound on the runtime of the distribution algorithm for a value n is $d(n)$, the *total* number of divisors of n, regardless of balance. For even without early cutoffs owing to range limitations, the algorithm cannot examine, given

$$n = \Pi_{i=1}^{i=k} p_i^{\alpha_i}$$

more than

$$d(n) = \Pi_{i=1}^{i=k} (1 + \alpha_i)^n$$

possibilities, since each factor $p_i^{\alpha_i}$ can, without range cut-offs, be distributed in $1 + \alpha_i$ ways.

It is well known that the average number of divisors $d(n)$ of a number near n is approximately $d(n) = \ln(n)$. This can easily be derived using the same sort of argument as we used above for balanced products [2]. This suggests that on average, the distribution process will not have many cases to examine; even for quad precision, we have $n \leq 2^{230}$ and so $\ln(n) \leq 160$.

What about the worst case? The number of divisors of a number can be much larger than $\ln(n)$. In fact [2], *almost all* numbers (in a precise sense) have about $\ln(n)^{\ln(2)}$ divisors, with the larger overall average of $\ln(n)$ resulting from a small proportion of numbers with many more divisors. Asymptotically, it is known [2] that $d(n)$ has an upper limit of exactly $2^{\ln(n)/\ln(\ln(n))}$, or more precisely, that if $\epsilon > 0$ then $d(n) < 2^{(1+\epsilon)\ln(n)/\ln(\ln(n))}$ for all sufficiently large n, while $d(n) > 2^{(1-\epsilon)\ln(n)/\ln(\ln(n))}$ for infinitely many n.

This asymptotic limit needs refinement to be useful to us for the concrete ranges we are interested in. We can obtain a more refined estimate of the maximum $d(n)$ for all n below some limit N we are interested in as follows. The key to efficient search is to seek the *minimal* n with the *maximal* number of divisors possible for $n \leq N$. The minimality constraint forces strong patterns onto the prime factorization. Suppose that n has the following prime factorization:

$$n = \Pi_{i=1}^{i=k} p_i^{\alpha_i}$$

Let $p_i < p_j$ be two primes (not necessarily appearing with nonzero index in the above factorization) such that $p_i^{\beta} < p_j < p_i^{\beta+1}$ for some nonnegative integer β. Then it is easy to see that if n has the minimality property, the following relationships hold between the α's:

$$\beta\alpha_j \leq \alpha_i \leq (\beta+1)\alpha_j + 2\beta$$

For if the first inequality failed we could get a smaller number with at least as many divisors by replacing $p_i^{\alpha_i} p_j^{\alpha_j}$ with $p_i^{\alpha_i+\beta} p_j^{\alpha_j-1}$, while if the second inequality failed we could likewise replace it with $p_i^{\alpha_i-(\beta+1)} p_j^{\alpha_j+1}$.

This observation includes the case where p_j is the first prime beyond those appearing in the factorization, and in this case $\alpha_i \leq 2\beta$. For example, if 17^{α} appears in the factorization, so must $3^{2\alpha}$ and $2^{4\alpha}$, while if no power of 17 appears in the factorization then the highest possible power of 2 appearing is 2^8, and the highest power of 3 is 3^6. Note in particular that the factorization of the minimal n must contain the first k consecutive primes without gaps, for some k.

These observations cut down the search space dramatically enough that we can easily perform an exhaustive search for the precise worst numbers up to quite large values, say 2^{3000}. The following table shows, for various values of p up to 230, the minimal $n \leq 2^p$ with the largest number of divisors possible in that range. For each such n, we show $\log_2(n)$ and $\log_2(d(n))$ (where $d(n)$ is the number of divisors of n), as well as the ratio with the expected limit superior $r(n) = \log_2(d(n))/(\ln(n)/\ln(\ln(n)))$ and the actual factorization of n.

p	$\log_2(n)$	$\log_2(d(n))$	$r(n)$	Factorization of that worst n
10	9.71	5.00	1.416	$2^3\ 3\ 5\ 7$
20	19.45	7.90	1.525	$2^4\ 3^2\ 5 \cdots 13$
30	29.45	10.39	1.535	$2^6\ 3^3\ 5^2\ 7 \cdots 17$
40	39.80	12.71	1.528	$2^6\ 3^4\ 5^2\ 7 \cdots 23$
50	49.84	14.75	1.512	$2^5\ 3^3\ 5^2\ 7^2\ 11 \cdots 31$
60	59.96	16.71	1.498	$2^6\ 3^4\ 5^3\ 7^2\ 11 \cdots 37$
70	69.42	18.49	1.488	$2^7\ 3^4\ 5^2\ 7^2\ 11 \cdots 43$
80	79.88	20.33	1.474	$2^8\ 3^5\ 5^3\ 7^2\ 11 \cdots 47$
90	89.90	22.07	1.463	$2^8\ 3^4\ 5^3\ 7^2\ 11 \cdots 59$
100	99.88	23.75	1.453	$2^7\ 3^5\ 5^3\ 7^2\ 11^2\ 13 \cdots 61$
110	109.64	25.33	1.443	$2^8\ 3^5\ 5^3\ 7^2\ 11 \cdots 71$
120	119.87	26.97	1.435	$2^7\ 3^6\ 5^3\ 7^2\ 11^2\ 13 \cdots 73$
130	129.87	28.56	1.427	$2^7\ 3^6\ 5^3\ 7^2\ 11^2\ 13^2\ 17 \cdots 79$
140	139.99	30.12	1.420	$2^{10}\ 3^5\ 5^4\ 7^2\ 11^2\ 13^2\ 17 \cdots 83$
150	149.74	31.66	1.416	$2^9\ 3^5\ 5^3\ 7^2\ 11^2\ 13^2\ 17 \cdots 97$
160	159.79	33.14	1.408	$2^8\ 3^6\ 5^3\ 7^3\ 11^2\ 13^2\ 17 \cdots 101$
170	169.83	34.66	1.404	$2^9\ 3^5\ 5^3\ 7^2\ 11^2\ 13^2\ 17 \cdots 107$
180	179.99	36.14	1.398	$2^8\ 3^6\ 5^3\ 7^3\ 11^2\ 13^2\ 17 \cdots 109$
190	189.82	37.56	1.393	$2^9\ 3^5\ 5^4\ 7^2\ 11^2\ 13^2\ 17^2\ 19 \cdots 113$
200	199.88	39.02	1.388	$2^{10}\ 3^6\ 5^3\ 7^3\ 11^2\ 13^2\ 17^2\ 19 \cdots 127$
210	209.93	40.43	1.383	$2^{10}\ 3^6\ 5^3\ 7^3\ 11^2\ 13^2\ 17 \cdots 137$
220	219.87	41.83	1.379	$2^8\ 3^5\ 5^4\ 7^3\ 11^2\ 13^2\ 17^2\ 19 \cdots 139$
230	229.92	43.21	1.375	$2^{10}\ 3^5\ 5^3\ 7^3\ 11^2\ 13^2\ 17 \cdots 151$

We can see that even for double-extended precision, the number of factorizations that could possibly need to be examined is about 2^{28}. Although a fairly large number, this is definitely feasible. (And of course in practice such cases are exceptional and not all factorizations would be examined.) For quad precision, on the other hand, it is entirely possible for the search to be infeasible. We have not yet encountered this phenomenon in practice, however.

Note that $d(n)$ also gives an upper bound to the number of balanced factorizations. It is, of course, pessimistic, but testing on some of the values above suggests that the the number of balanced factorizations is a reasonable proportion (say 10%) of the total number of divisors. Naturally, it would be better to refine all these estimates to consider only numbers very close to the powers of 2, which is what we are really interested in.

The special numbers that we searched for above are particular cases of *highly composite numbers* [13]. For a detailed survey of the subject see [10], while Achim Flammenkamp's Web page seems to give a more efficient algorithm for generating HCNs:

```
http://wwwhomes.uni-bielefeld.de/achim/highly.html
```

The sequence of highly composite numbers is A002182 in Sloane's Encyclopedia of Integer Sequences.

8 Extension to reciprocal square root

It is interesting to note that a similar factor distribution technique can be used to attempt to find exceptional cases for the reciprocal square root. In this case, we seek floating-point numbers or midpoints w and floating-point numbers y such that

$$\frac{|w - \frac{1}{\sqrt{y}}|}{|\frac{1}{\sqrt{y}}|}$$

is small. We can rewrite this as:

$$|\sqrt{y}(w - \frac{1}{\sqrt{y}})| = |w\sqrt{y} - 1|$$

In the critical cases where $w\sqrt{y} - 1$ is very small, then $w\sqrt{y} + 1$ is almost exactly 2 and so:

$$|w\sqrt{y} - 1| = \frac{|w^2 y - 1|}{|w\sqrt{y} + 1|} \approx \frac{|w^2 y - 1|}{2}$$

Once again, let us scale the values w and y to integers m and b:

$$\begin{array}{ll} y = 2^{e_y} b & 2^{p-1} \leq b < 2^p \\ w = 2^{e_w} m & 2^p \leq m < 2^{p+1} \end{array}$$

and then the distance we are interested in is then:

$$\frac{|m^2 b - 2^q|}{2^{q+1}}$$

where $q = -(2e_w + e_y)$. So we seek cases where $d = m^2 b - 2^q$ is as small as possible. Keeping in mind the range restrictions, we see that $2^{3p-1} \leq m^2 b < 2^{3p+2}$. As with simple reciprocals, it is impossible to come very close to the extremal powers of 2, but we do now need to consider two cases, $q = 3p$ and $q = 3p + 1$.

The reciprocal square root function is of some theoretical interest because it seems *prima facie* possible that $d = m^2 b - 2^q$ could be very small, perhaps even ± 1, yet no precisions where it is much smaller than 2^p have ever been found, and one might expect on naive statistical grounds that it is unlikely. (We only have 2^{2p} different choices of pairs m and b, and are scattering the resulting $m^2 b$'s somehow over an interval of size about 2^{3p}.) Li [7] proves that *assuming* the ABC conjecture from number theory holds, the distance is indeed of order 2^p for all sufficiently large p. Even if the ABC conjecture were proven, however, it's not clear whether it would be possible to constructivize the proof in order to obtain useful bounds for specific precisions. Iordache and Matula [4] observe that $d = 1$ is impossible in general, allowing the accuracy required to be lowered slightly, but add that 'trying to lower it is not an easy problem, even for a fixed p'. Although the present work does not touch the general case, and nor can it fully bridge the gap between expected and provable bounds, it *does* allow us quite easily to improve the provable bound for the typical p we are interested in by a reasonable factor.

We can take over the prime distribution function with little change. The only difference is that we now need to distribute the prime factors among $m^2 b$. This has the immediate consequence that only even powers of primes can be allocated to the m^2 part, and so any prime appearing to an odd power in the prime factorization of $2^q + d$ must be allocated at least once to b. This is almost always enough to render the distribution immediately impossible. We have made some searches for double-extended precision ($p = 64$) and quad precision ($p = 113$). For double-extended, we have shown that $d \leq 1024$ is impossible, and it would be easy to continue the search much further. For quad precision, the cost of factoring numbers is now a serious bottleneck, with a single number sometimes taking a day of CPU time and one of the factorizations for the $d = 6$ case apparently defeating factorization in a reasonable time. Nevertheless we have at least shown that $d < 6$ is impossible, which represents some improvement. For smaller precisions, it seems likely that other algorithms based on an (intelligent) exhaustive analysis of the whole space of significands would be more efficient. For example Lang and Muller [5] have performed a complete analysis of the double-precision case $p = 53$ (and found that the minimal distance is about 2^{-110}).

9 Conclusion

The methods described here allow reasonably effective isolation of the 'worst cases' for the reciprocal function. This opens the way to correctness proofs of reciprocal algorithms using the same kind of two-part approach used by Cornea [1] for square roots. In the absence of new theoretical advances, the method described may also be the best available means of improving the difficulty bounds on the reciprocal square root functions for larger precisions. Although our method has feasibility problems for the extreme case of quad-precision reciprocal square roots, it would be possible to explore alternative factoring algorithms. The numbers we are interested in factoring are very close (in relative terms) to powers of 2, so it is possible that algorithms such as the Special Number Field Sieve (SNFS) would give much better results.

Acknowledgements

The author is grateful to the anonymous referees, who made a number of excellent suggestions, and pointed out connections of which the author had been unaware.

References

[1] M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide

and remainder algorithms. *Intel Technology Journal*, 1998-Q2:1–11, 1998. Available on the Web as `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.

[2] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Clarendon Press, 5th edition, 1979.

[3] J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 2000.

[4] C. Iordache and D. W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In I. Koren and P. Kornerup, editors, *Proceedings, 14th IEEE symposium on on computer arithmetic*, pages 233–240, Adelaide, Australia, 1999. IEEE Computer Society. See also Technical Report 99-CSE-1, Southern Methodist University.

[5] T. Lang and J.-M. Muller. Bounds on runs of zeros and ones for algebraic functions. Research Report 4045, INRIA, 2000.

[6] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. Research Report 4044, INRIA, 2000.

[7] R.-C. Li. The ABC conjecture and correctly rounded reciprocal square root. Preprint, 2002.

[8] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Prentice-Hall, 2000.

[9] P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM Journal of Research and Development*, 34:111–119, 1990.

[10] J.-L. Nicholas. On highly composite numbers. In *Ramanujan Revisited: Proceedings of the Centenery Conference*, pages 215–244. Academic Press, 1988.

[11] M. Parks. Number-theoretic test generation for directed rounding. *IEEE Transactions on Computers*, 49:651–658, 2000.

[12] V. Pratt. Every prime has a succinct certificate. *SIAM Journal of Computing*, 4:214–220, 1975.

[13] S. Ramanujan. Highly composite numbers. *Proceedings of the London Mathematical Society*, 14:347–409, 1915.

[14] P. T. P. Tang. Testing computer arithmetic by elementary number theory. Preprint MCS-P84-0889, Mathematics and Computer Science Division, Argonne National Labs, 1989.