Kahan's algorithm for a correct discriminant computation at last formally proven

Sylvie Boldo

Kahan's Algorithm for a Correct Discriminant Computation at Last Formally Proven

Sylvie Boldo

Abstract—This article tackles Kahan's algorithm to compute accurately the discriminant. This is a known difficult problem, and this algorithm leads to an error bounded by 2 ulps of the floating-point result. The proofs involved are long and tricky and even trickier than expected as the test involved may give a result different from the result of the same test without rounding. We give here the total demonstration of the validity of this algorithm, and we provide sufficient conditions to guarantee that neither overflow nor underflow will jeopardize the result. The IEEE-754 double-precision program is annotated using the Why platform and the proof obligations are done using the Coq automatic proof checker.

Index Terms—Floating point, discriminant, formal proof, Why platform, Coq.

1 Introduction

FLOATING-POINT arithmetic is a field that seems strange and obscure to most programmers. However, floating-point experts can create very tricky algorithms that take advantage of inexact but well-specified computation to get correct, accurate, or consistent results.

This category includes, for example

- expansion algorithms that allow multiprecision computations using only the floating-point unit [1], [2],
- CRlibm that computes correctly rounded elementary functions on IEEE double precision [3],
- multiprecision algorithms on higher precision with correct rounding [4],
- Horner's rule under assumptions (such as elementary function evaluation) [5],
- accurate summation under tough assumptions (all numbers are nonnegative, for example) [6], and
- accurate discriminant computation [7].

We here are interested in the last algorithm. The reason is that the pen-and-paper proofs provided are described as "far longer and trickier" than the algorithms and programs and Kahan deferred their publication [7].

Due to the difficulty of the proof, we are interested in formally proving this algorithm. Moreover, we want to prove it fully, with overflow and underflow included. The application range of the algorithm may be too tight, but we want to be able to give sufficient conditions for this algorithm to work as expected, meaning to give an accurate result.

This program (Program 1) computes the value of the discriminant, meaning that given a, b, and c, it computes $b^2 - ac$.

This value is of course usually not the final answer of a program, but it is used in many applications. For example, the orientation test determines whether a point lies to the left of, to the right of, or on a line or plane defined by other points. To know if a point p defined by its abscissa and its ordinate lies on the left or on the right of $\overrightarrow{qr}$, we compute

$$\begin{vmatrix} q_x & q_y & 1 \\ r_x & r_y & 1 \\ p_x & p_y & 1 \end{vmatrix} = \begin{vmatrix} q_x - p_x & q_y - p_y \\ r_x - p_x & r_y - p_y \end{vmatrix}.$$

Its sign gives the answer. And this orientation test is known to possibly give an incorrect answer due to round-off errors [8]. This is of course more complex than a discriminant as there may be initial errors on a, b, and c, but this means that the error of the discriminant computation must be under control.

Another example is a simple linear system such as a mechanical system with a mass, a spring, and a drag or an electrical system with an inductance, a capacitance, and a resistance. After disturbance by an impulse, knowing if this system passes through its equilibrium state before returning to it can be answered by determining whether the roots of a degree-2 polynomial are real or complex, and the roots tell how long the system takes to return to equilibrium [7].

The responsibility of the discriminant program is then rather high. This calls for a high level of reliability in the proof of the correctness of this program. This is the reason why we guarantee it using formal proofs. The formalization of the floating-point numbers and arithmetic is the following one [9]: a float is a pair of signed integers (n, e) with both n and e bounded and has a value equal to $n \times 2^e$. This formalization has permitted proving both old and new results [10].

As we want to prove the real program, we use the Why platform and the Caduceus tool for the verification of C programs [11], [12] and the associated floating-point annotations [13]. There is therefore no doubt about the algorithm proved and the property proved because the given specifications are C-like and, therefore, much more understandable than Coq theorems.

Notations. All floating-point numbers are in IEEE double-precision format. For the sake of simplicity, we denote as $prec$ the precision, so $prec = 53$. We denote by μ the smallest (subnormal) positive floating-point number, so $\mu = 2^{-1,074}$. All roundings are therefore to nearest, ties to even and are denoted by $\circ$.

The unit in the last place is denoted by ulp. For a floating-point number, it is the IEEE-754 definition: the value of the last bit of the significand. For example, $\mathrm{ulp}(1) = 2^{-52}$, $\mathrm{ulp}(2^k) = 2^{k-52}$, and $\mathrm{ulp}(0) = \mu$ (like for any subnormal). For a real number, there are various possible definitions [14] with different properties. We use the floating-point ulp unless stated otherwise. There is a discussion about the use of the ulp of the exact real result in Section 8.

The successor of a float x is denoted as x^+, and its predecessor is denoted by x^-.

The model for a floating-point number x is composed of an integer significand n_x such that $|n_x| < 2^{prec}$ and of an exponent e_x. The real value associated to x is then $n_x \times 2^{e_x}$. Overflow problems will be tackled in Section 5 and underflow in Section 6. Until then, we assume an unbounded exponent range.

2 The Program

The initial program is an incredibly long program in Matlab [7] that handles any kind of floating-point arithmetic (after testing the underlying architecture). We assume here that we use IEEE-754 double-precision floating-point arithmetic. This means that we know beforehand the radix (2), the precision (53), and the rounding (to nearest, ties to even; we assume that the programmer did not change it). We then rewrite the initial program in the C language (Program 1).

• *The author is with the INRIA Saclay—Île-de-France, Parc Orsay Université—ZAC des Vignes, 4 rue Jacques Monod—Bâtiment N, 91893 ORSAY Cedex, France. E-mail: sylvie.boldo@inria.fr.*

Manuscript received 20 July 2007; revised 7 May 2008; accepted 18 Sept. 2008; published online 23 Oct. 2008.
Recommended for acceptance by P. Kornerup, P. Montuschi, J.-M. Muller, and E. Schwarz.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-2007-07-0334.
Digital Object Identifier no. 10.1109/TC.2008.200.

Program 1 Accurate discriminant computation

```
double Kahan_discr
       (double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=exactmult(b,b,p);
    dq=exactmult(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

The functions used inside `Kahan_discr` are the following:

- `fabs`. This function computes the absolute value of a floating-point number.
- `exactmult`. This function computes the floating-point error of a multiplication. Indeed, for two floating-point numbers x and y, if xy is the rounded result of $x \times y$, then the value $xy - x \times y$ fits in a floating-point number and can be effectively computed using floating-point operations, provided that there is neither overflow nor underflow. This algorithm was discovered at the same period by Veltkamp and Dekker [1], [15], [16], and the respective paternities are unknown.

If a fused multiply-and-add is available, a very simplified algorithm gives the same result as `exactmult` using only one floating-point operation [17].

The function `fabs` is assumed to give the correct answer (that is to say, the float whose value is the absolute value of the input). The function `exactmult` is proved to be correct with underflow conditions. The proof of this algorithm is described in [18] for any radix. The program given here does not rely on any other complex function: the demonstration is self-contained.

3 Proof of the "Ideal" Algorithm

This partial proof has already been published in [19]. The main ideas of the proof are nevertheless explained, as there was not space enough in [19] and as part of them will be used afterward. Moreover, the error bound was incorrectly stated in [19]: the ulps stated there also were the floating-point ulps and not the ulps of the real exact value. The notations are those of Program 1.

The "ideal" algorithm means that we assume that the test is made on real values: the actual test is `if` $\circ(p+q) \leq \circ(3 \circ (|p-q|))$, but we here assume that we use `if` $p+q \leq 3 \times |p-q|$. The limits of this approach will be discussed in the next section.

We assume for this section that there is neither underflow nor overflow. Let us prove that $b^2 - ac$ is within 2 ulps of d. Let $\delta = |d - (b^2 - ac)|$, we will prove that $\delta \leq 2\mathrm{ulp}(d)$.

We know that $p \geq 0$ as $p = \circ(b^2)$.

3.1 First Case: $3|p-q| \geq p+q$

This corresponds to the first possibility in the `if`. We here split this case into two subcases depending on the respective values of p and q.

3.1.1 *First Subcase:* $p \geq q$

Then, $|p-q| = p-q$, and we know that $3(p-q) \geq p+q$; therefore, $p \geq 2q$, and $p - q \geq \frac{p}{2}$.

- Assume that q is positive or zero.
  If $q \geq 0$, then $p \geq 2q$ implies that $\mathrm{ulp}(p) \geq 2\mathrm{ulp}(q)$. As $p - q \geq \frac{p}{2} \geq 0$, we have the fact that $\mathrm{ulp}(d) \geq \frac{1}{2}\mathrm{ulp}(p)$. Then, $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(p) + \frac{1}{2}\mathrm{ulp}(q) \leq 2\mathrm{ulp}(d)$.
- Assume that q is negative.
  Then, p and $-q$ share the same sign (or p is zero), $d \geq p$, and $d \geq |q|$, so we easily deduce that

$$\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(p) + \frac{1}{2}\mathrm{ulp}(q) \leq \frac{3}{2}\mathrm{ulp}(d).$$

3.1.2 *Second Subcase:* $q \geq p$

Then, $q \geq 0$, $|p-q| = -p+q$, and $3(-p+q) \geq p+q$; therefore, $q \geq 2p$, and $q - p \geq \frac{q}{2} \geq 0$. Then, $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(p) + \frac{1}{2}\mathrm{ulp}(q)$, and therefore, $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(d) + \mathrm{ulp}(d) = 2\mathrm{ulp}(d)$.

3.2 Second Case: $3|p-q| < p+q$

This corresponds to the second possibility in the `if`. This case is much more complex. We now have more intermediate values for the multiplication errors: $p + dp = b^2$ and $q + dq = ac$. We prove several intermediate results before splitting it into cases. First, if $p = q$, then d is correct within half an ulp as $d = \circ(b^2 - ac)$. Let us now assume that $p \neq q$.

Lemma 1. *The computation of $p - q$ is exact.*

Proof. If $q \leq 0$, then we have $3|p-q| < p+q = |p| - |q| \leq |p-q|$, which is absurd. Therefore, $q > 0$. Thus, $p = |p| \leq |p-q| + |q| \leq \frac{1}{3}(p+q) + q$, so $p \leq 2q$.

In the same way, $q = |q| \leq |p-q| + |p| \leq \frac{1}{3}(p+q) + p$, so $q \leq 2p$. Therefore, the subtraction is exact by Sterbenz's theorem [20]. □

Lemma 2. $|\circ(dp - dq)| \leq 2|p-q|$.

Proof. As $p \neq q$, we know the fact that $|p-q| \geq \min(\mathrm{ulp}(p), \mathrm{ulp}(q))$. We also know that $\max(\mathrm{ulp}(p), \mathrm{ulp}(q)) \leq 2\min(\mathrm{ulp}(p), \mathrm{ulp}(q))$ as $\frac{q}{2} \leq p \leq 2q$:

$$\begin{aligned} |dp - dq| &\leq |dp| + |dq| \leq \frac{1}{2}\mathrm{ulp}(p) + \frac{1}{2}\mathrm{ulp}(q) \\ &\leq \frac{3}{2}\min(\mathrm{ulp}(p), \mathrm{ulp}(q)) \leq \frac{3}{2}|p-q|. \end{aligned}$$

Therefore, $|\circ(dp - dq)| \leq 2|p-q|$. □

This may seem a not-tight-enough bound, but we are in the few cases where $p \approx q$. Therefore, $p - q$ may be very small, so this bound ($2 \times |p-q|$) is not far from the lowest possible bound (just under 1.5) among the bounds of the type $k \times |p-q|$.

Lemma 3. *If the computation of $dp - dq$ is exact, then $\delta \leq 2\mathrm{ulp}(d)$.*

Proof. If the computation of $dp - dq$ is exact, then we may simplify δ by $\delta = |\circ(b^2 - a \times c) - (b^2 - a \times c)| \leq \frac{1}{2}\mathrm{ulp}(d)$. □

We will now evaluate δ in all possible cases.
The first easy case is when $\mathrm{ulp}(p) = \mathrm{ulp}(q)$.

Lemma 4. *When $\mathrm{ulp}(p) = \mathrm{ulp}(q)$, then $\delta \leq 2\mathrm{ulp}(d)$.*

Proof. We prove that the computation of $dp - dq$ is exact, so the result holds by Lemma 3. Let e be such that $2^e = \mathrm{ulp}(p) = \mathrm{ulp}(q)$.

As $dp = b^2 - \circ(b^2)$, we know that dp fits in $prec$ bits with exponent $e - prec$. Similarly, $dq = a \times c - \circ(a \times c)$, and dq fits in $prec$ bits with the same exponent $e - prec$. The mantissa corresponding to $dp - dq$ with exponent $e - prec$ is then an integer m such that $|m| = |dp - dq| 2^{-e+prec} \leq (|dp| + |dq|) 2^{-e+prec} \leq 2^{prec}$. Either we have $|m| = 2^{prec}$, which implies that $|dp - dq|$ is equal to 2^e, so $dp - dq$ is computed exactly or we have $|m| < 2^{prec}$, which implies that $dp - dq$ fits in $prec$ bits with the chosen exponent and is therefore computed exactly. □

We now split the other cases into two subcases.

3.2.1 General Case $|p-q| \geq 3\min(\mathrm{ulp}(p), \mathrm{ulp}(q))$

Then, $|dp-dq| \leq \frac{3}{2}\min(\mathrm{ulp}(p), \mathrm{ulp}(q)) \leq \frac{1}{2}|p-q|$. So $|\circ(dp-dq)| \leq \frac{1}{2}|p-q|$ as $p-q$ is computed correctly by using Lemma 1. Moreover, $|p-q+\circ(dp-dq)| \geq |p-q| - |\circ(dp-dq)| \geq \frac{1}{2}|p-q| \geq |dp-dq|$. So by the monotonicity of ulp and $\circ$, we have $\mathrm{ulp}(\circ(dp-dq)) \leq \mathrm{ulp}(\circ(p-q+\circ(dp-dq))) = \mathrm{ulp}(d)$. Therefore, in that case, we have $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(\circ(dp-dq)) \leq \mathrm{ulp}(d)$.

3.2.2 Particular Case

Now, we assume that we are not in the general case. Therefore, we have left the cases where the ulps of p and q are different but $|p-q|$ is either $2\min(\mathrm{ulp}(p), \mathrm{ulp}(q))$ or $\min(\mathrm{ulp}(p), \mathrm{ulp}(q))$. It means that p and q are very near a power of 2.

As p and q are symmetrical here, we assume without loss of generality that $p \geq q$. We then shift the exponents of p and q so that they are near 1.

We use the following notations: x^+ is the successor of x, and x^- is its predecessor. The possible cases are then either $(p=1, q=1^-)$ with $p-q=\mathrm{ulp}(q)$, $(p=1, q=1^{--})$ with $p-q=2\mathrm{ulp}(q)$, or $(p=1^+, q=1^-)$ with $p-q=3\mathrm{ulp}(q)$. Note that the latest case fits into the general case.

We now assume that $p=1$ and q is either 1^- or 1^{--}:

a. Assume that dp and dq share the same sign.
Then, we prove that the computation of $dp-dq$ is exact, which implies the result by Lemma 3.
If $dq=0$, then $dp-dq$ is exactly dp. If not, then we know many things about dp and dq as $p=1$ and $q=1^-$ or 1^{--}. It implies that dq fits in $prec$ bits with exponent $-2prec$ and that dp fits in $prec$ bits with exponent $-2prec+1$. So $dp-dq$ can use the exponent $-2prec$. Moreover, as $|dp| \leq 2^{-prec}$, $|dp-dq| < 2^{-prec}$, so $|dp-dq| \leq 2^{-prec} - 2^{-2prec}$ that fits in $prec$ bits. So $dp-dq$ is computed exactly.

b. Let us now assume that dp and dq have opposite signs. We split one more time into two subcases:

   i. Assume that $dp-dq \geq 0$.
   Then, $d \geq p-q = 2^{-prec}$ if $q=1^-$ and $d \geq 2^{1-prec}$ if $q=1^{--}$.
   In any case, $dp-dq \leq 2^{-prec} + 2^{-1-prec} = 3\times 2^{-1-prec}$.
   So $\circ(dp-dq) \leq 3\times 2^{-1-prec}$, and we have the fact that $\mathrm{ulp}(\circ(dp-dq)) \leq 2^{1-2prec}$.
   Finally, we bound $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(\circ(dp-dq))$.
   Then, $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + 2^{-2prec} \leq \frac{1}{2}\mathrm{ulp}(d) + 2^{-prec}d$, and $\delta \leq \frac{3}{2}\mathrm{ulp}(d)$.

   ii. Assume that $dp-dq \leq 0$.
   It means that $dp \leq 0$. As $p=1$, we easily have that $|dp| \leq 2^{-1-prec}$. We then prove that $dp-dq$ is computed exactly, and this implies the result by Lemma 3.
   As dp and dq accept $-2prec$ as exponent and also as $|dp-dq| \leq 2^{-1-prec} + 2^{-1-prec} = 2^{-prec}$, either we have $|dp| = |dq| = 2^{-1-prec}$, and in this case, $dp-dq = -2^{-prec}$ is indeed representable or we have $|dp-dq| < 2^{-prec}$, so it fits in $prec$ bits with exponent $-2prec$.

In all subcases of all cases, the rounding error is bounded, and the result holds.

4 Proof of the Program

We now prove that the real-life program satisfies the same property, meaning that b^2-ac is within 2 ulps of d. Let us now assume a floating-point test. Unfortunately, this means that the preceding proof does not work any more. There may indeed be cases where the real test and the floating-point test disagree.

Let us use a toy-example floating-point format with a five-digit significand. Let $p = 27 = 11011_2$ and $q = 14 = 1110_2$; then, $p+q = 41 > 3|p-q| = 3\times 13 = 39$. But $\circ(p+q) = \circ(41) = 40$, $\circ(p-q) = p-q = 13$, and $\circ(3\circ(p-q)) = \circ(39) = 40$. Therefore, we have

$$\circ(p+q) = \circ(3\circ(p-q)) \text{ and } p+q > 3|p-q|,$$

so the floating-point and real tests disagree.

In most cases, the floating-point and real tests agree; then, the result holds by the results of Section 3. We now just have to look into disagreements.

4.1 First Disagreement

We first assume that $3|p-q| < p+q$ and, on the "contrary," that $\circ(3\circ(|p-q|)) \geq \circ(p+q)$. We may assume without loss of generality that $p \geq q$.

Lemma 5. *If $e_d \geq e_q$, then $\delta \leq 2\mathrm{ulp}(d)$.*

Proof. As $3|p-q| < p+q$, we can apply Lemma 1 to prove that $p-q$ is computed exactly, so $d = p-q$. Then

$$\delta = |p-q-(b^2-ac)| \leq \frac{1}{2}(\mathrm{ulp}(p)+\mathrm{ulp}(q)) \leq \frac{3}{2}\mathrm{ulp}(q) \leq \frac{3}{2}\mathrm{ulp}(d).$$

□

We now assume without loss of generality that $e_d < e_q$.

We will first prove that p and q are very near one to the half of the other.

Lemma 6. $\frac{p}{2} \leq q \leq p\frac{1+2^{1-prec}}{2+2^{-prec}}$.

Proof. As $3|p-q| < p+q$, we know that p and q fit in Sterbenz's theorem. Therefore, $q \leq p \leq 2q$ and $\circ(p-q) = p-q$ by Lemma 1. So we know that

$$\circ(p+q) \leq \circ(3\circ(|p-q|)) = \circ(3|p-q|) = \circ(3(p-q)).$$

In fact, we have $p \approx 2q$:

$$\begin{aligned} p+q-3(p-q) &\leq (p+q) - \circ(p+q) + \circ(p+q) - \circ(3(p-q)) \\ &\quad + \circ(3(p-q)) - 3(p-q) \\ &\leq \frac{1}{2}\mathrm{ulp}(\circ(p+q)) + 0 + \frac{1}{2}\mathrm{ulp}(\circ(3(p-q))) \\ &\leq \mathrm{ulp}(\circ(3(p-q))) \leq 2^{1-prec}\circ(3(p-q)) \\ &\leq 2^{1-prec}\times 3(p-q)\frac{1}{1-2^{-prec}}. \end{aligned}$$

So $-2p+4q \leq (3p-3q)\frac{2^{1-prec}}{1-2^{-prec}}$, and therefore, $q \leq p\frac{1+2^{1-prec}}{2+2^{-prec}}$. □

As p and q are positive, $p \geq q$ implies that either:

$$\mathrm{ulp}(p) = \mathrm{ulp}(q) \text{ or } \mathrm{ulp}(p) > \mathrm{ulp}(q).$$

Lemma 7. *If $\mathrm{ulp}(p) = \mathrm{ulp}(q)$, then $\delta \leq 2\mathrm{ulp}(d)$.*

Proof. If $\mathrm{ulp}(p) = \mathrm{ulp}(q)$, then $\delta \leq \frac{1}{2}(\mathrm{ulp}(p)+\mathrm{ulp}(q)) = \mathrm{ulp}(q)$. And $d = p-q \geq q\frac{1-2^{-prec}}{1+2^{1-prec}} \geq \frac{q}{2}$ by Lemma 6, so $\mathrm{ulp}(d) \geq \frac{\mathrm{ulp}(q)}{2}$, and $\delta \leq 2\mathrm{ulp}(d)$. □

Lemma 8. *If $\mathrm{ulp}(p) > \mathrm{ulp}(q)$, then $2q = p^+$.*

Proof. As $3|p-q| < p+q$ and $p \geq q$, one has $p < 2q$ and $p^+ \leq 2q$.

Next, $p^{++} \geq p + 2\mathrm{ulp}(p) \geq p + \frac{2p}{2^{prec}-1} = p\frac{2^{prec}+1}{2^{prec}-1}$. So after a few computations, we have $p^{++} \geq p\frac{2^{prec}+1}{2^{prec}-1} > p\frac{1+2^{1-prec}}{1+2^{-1-prec}} \geq 2q$, so $p^+ \geq 2q$. □

Lemma 9. *$\mathrm{ulp}(p) > \mathrm{ulp}(q)$ is impossible.*

Proof. This is achieved by proving that the equality $2q = p^+$ of Lemma 8 is impossible. We look at all the possible cases for the significand n_q:

- If q is a power of 2, then as $p^+ \geq 2q$, we have $p = (2q)^-$, so $\mathrm{ulp}(p) = \mathrm{ulp}((2q)^-) = \mathrm{ulp}(2q)/2 = \mathrm{ulp}(q)$. This is impossible in this subcase.
- If q is not a power of the radix, then $p = (2q)^-$ is equal to $2q - 2\mathrm{ulp}(q)$. As $e_d \leq e_q - 1$, we have $d < 2^{prec-1+e_q}$, so $2^{prec-1+e_q} > d = p - q = q - 2\mathrm{ulp}(q) = (n_q - 2)2^{e_q}$, and therefore, $n_q < 2^{prec-1} + 2$. The facts that q can be normalized and is not a power of 2 imply that $n_q = 2^{prec-1} + 1$. So $p = 2^{prec+e_q}$.

  Then, $3|p-q| = 2^{e_q} \times 3 \times (2^{prec} - (2^{prec-1}+1))$, so $3|p-q| = 2^{e_q}(3 \times 2^{prec-1} - 3)$ is rounded into $2^{e_q}(3 \times 2^{prec-1} - 4)$. And $p + q = 2^{e_q}(2^{prec} + 2^{prec-1} + 1)$ is rounded into the value $2^{e_q}(3 \times 2^{prec-1})$. Those values are not equal as they should. □

By Lemmas 5, 7, 8, and 9, in all possible cases, we have the fact that $\delta \leq 2\mathrm{ulp}(d)$.

4.2 Second Disagreement

We now assume that $3|p-q| \geq p+q$ and, on the "contrary," that $\circ(3 \circ (|p-q|)) < \circ(p+q)$. We may assume without loss of generality that $p \geq q$.

Lemma 10. $p - q \leq \frac{1}{3}(p + |q|)\frac{(1+2^{-prec})^2}{1-2^{-prec}}$.

Proof. We have

$$\begin{aligned} p - q &\leq \circ\,(p-q)(1+2^{-prec}) \\ &\leq \frac{1}{3} \circ (3 \circ (|p-q|))(1+2^{-prec})^2 \\ &\leq \frac{1}{3} \circ (p+q)(1+2^{-prec})^2 \\ &\leq \frac{1}{3}(p+|q|)\frac{(1+2^{-prec})^2}{1-2^{-prec}}. \end{aligned}$$

□

Lemma 11. $0 < q$.

Proof. If $q \leq 0$, the inequality of Lemma 10 becomes $p - q \leq \frac{1}{3}(p-q)\frac{(1+2^{-prec})^2}{1-2^{-prec}}$, and that implies that $1 \leq \frac{1}{3}(1+\varepsilon)$ with $0 < \varepsilon \ll 1$, which is absurd. Therefore, $0 < q$. □

Lemma 12. $\circ(|dp - dq|) \leq 3\mathrm{ulp}(\circ(p-q))$.

Proof. As $3|p-q| \geq p+q$, we have $2q \leq p$ and $q \leq \circ(p-q)$.

Next, Lemmas 10 and 11 imply that $p - q \leq \frac{1}{3}(p+q)\frac{(1+2^{-prec})^2}{1-2^{-prec}}$; therefore, $p \leq 3q$.

So $|dp - dq| \leq \frac{1}{2}(\mathrm{ulp}(p) + \mathrm{ulp}(q)) \leq 3\mathrm{ulp}(q) \leq 3\mathrm{ulp}(\circ(p-q))$. So $\circ(|dp-dq|) \leq 3\mathrm{ulp}(\circ(p-q))$. □

Then, $\circ(p-q) + \circ(dp-dq) \geq \circ(p-q) - 3\mathrm{ulp}(\circ(p-q))$. Moreover, if f is representable, then $f - 3\mathrm{ulp}(f)$ is also representable. Therefore, $d \geq \circ(\circ(p-q) - 3\mathrm{ulp}(\circ(p-q)) = \circ(p-q) - 3\mathrm{ulp}(\circ(p-q))$, and $\mathrm{ulp}(d) \geq \frac{1}{2}\mathrm{ulp}(\circ(p-q))$.

Now, we can end the proof: $\delta \leq \frac{1}{2}(\mathrm{ulp}(d) + \mathrm{ulp}(\circ(p-q)) + \mathrm{ulp}(\circ(|dp-dq|)))$. And finally, $\delta \leq \mathrm{ulp}(d)(\frac{1}{2} + 1 + 2^{1-prec} \times 3 \times 2) \leq 2\mathrm{ulp}(d)$.

In any disagreement, the result still holds. Finally, the result holds whatever the results of the floating-point and real tests.

5 Overflow

Overflow is usually disregarded as it usually creates non-numerical quantities (NaN or $\pm\infty$). Unfortunately, it may happen that overflow occurs but that such quantities disappear during the following computations.

Our choice is to prohibit overflow: we prove that each and every computation has a result smaller than or equal to the maximal floating-point number in IEEE double precision, that is to say, $(2 - 2^{-52})2^{1,023}$. This is done using a command line option of the Why platform to turn on this check. As by default, we also prohibit division by zero and square root of negative numbers, this prevents any creation of infinities or NaN.

In Program 1 or in the `exactmult` function (see Program 2), you can easily get infinities or NaN as soon as any of these values gets bigger: either b^2 or $a \times c$, or even a, due to the $\circ((2^{27}+1) \times a)$.

We then require $|a|$ and $|c|$ to be smaller than 2^{995} so that $(2^{27}+1) \times a$ or $\times c$ does not overflow. We also require $|a \times c|$ to be smaller than $2^{1,020}$ and $|b| \leq 2^{510}$ so that the other computations, including especially $3 \times \circ(p-q)$, do not overflow. These assumptions are enough to guarantee that each and every computation will not overflow.

6 Underflow

The hypothesis "there is no underflow" is a customary one, usually used without a qualm. Nevertheless, it is really unclear, as a subnormal value is not difficult to get in the middle of a computation, even if all the inputs are normal.

One advantage of the use of the formal proof checker is that it forces us to add all the necessary hypotheses. Therefore, the "no-underflow" hypothesis must be clear so that the proof can be done. Moreover, these hypotheses can be worked on afterward. For example, we first assumed that the exponent of p must be greater than the minimal exponent plus 2. We then can try to prove that the minimal exponent plus 1 is a sufficient exponent. If we can prove it, then the theorem has been improved. The possibility of patching a proof afterward to improve it is a huge benefit.

The proved sufficient conditions for the preceding proof are the following:

- p, q, and d are either zeros or normal floats.
- $b^2 \geq 2^{2prec-1}\mu$ or $b = 0$.
- $|ac| \geq 2^{2prec-1}\mu$ or $ac = 0$.
- The exponents of d and $\circ(p-q)$ must be greater than the minimal exponent plus 1.

The idea is that we need all the involved floats to be normal so that the theorems used intuitively hold: for example, we need $\circ(r)$ to be normal to have $|r| \leq |\circ(r)|(1+2^{-prec})$. It includes dp and dq that must be either zeros or normal, hence the bounds on b^2 and $|ac|$.

These hypotheses are quite technical. The last one is especially ponderous as it depends on the exponent of a float and also as it depends on the value of either the output or an intermediary value. We replace the preceding conditions by these ones on the inputs that are sufficient to guarantee that there is no underflow:

- We have either $b = 0$ or $b^2 \geq 2^{3prec-1}\mu = 2^{-916}$.
- We have either $a \times c = 0$ or $|a \times c| \geq 2^{3prec-1}\mu = 2^{-916}$.

The formal proof checker helped us to think about all the subcases depending on the fact that a given float, namely, a, b, c, p, q, or d, is zero or not.

7 Fully Annotated Program

The fully annotated program is Program 2. The annotations interpreted by the Why platform are comments beginning with `@`. Details on the syntax of the Why platform's annotations can be found in [11], [12], and [13]. We here give the necessary keys to understand this program:

- Each function has some needed hypotheses to work correctly, beginning with `requires`. It also guarantees properties concerning its result, beginning with `ensures`.
- The result of a function is denoted by `\result`.
- `|x|` denotes the absolute value of x, and `2^^e` is 2^e.

- The notations && and || have the usual C meaning of $\wedge$ (and) and of $\vee$ (or).
- round(r) is the rounding to nearest even of the real r, and ulp is the unit in the last place of a floating-point number.

Note that the notation round is needed as all computations inside the annotations are exact ones. For example, x * y - xy means the exact real value $x \times y - xy$ with mathematical multiplication and subtraction and without any rounding.

Program 2 Fully annotated accurate discriminant computation

```
/*@ ensures \result==|f| */
double fabs(double f);

/*@ ensures \result==2^^e */
double exp2(int e);


/*@ requires xy==round(x*y) &&
  @    (x*y==0 || 2^^(-969) <= |x*y|) &&
  @    |x| <= 2^^995 && |y| <= 2^^995 &&
  @    |x*y| <=2^^1022
  @ ensures \result==x*y-xy
  @ */

double exactmult
       (double x, double y, double xy) {
  double C,px,hx,tx,py,hy,ty;
  C=exp2(27)+1;
  /*@ assert C==2^^(27)+1 */

  px=x*C;
  hx=(x-px)+px;
  tx=x-hx;

  py=y*C;
  hy=(y-py)+py;
  ty=y-hy;

  return -((((xy-hx*hy)-hx*ty)-hy*tx)-tx*ty);
}



/*@ requires
  @      (b==0   || 2^^(-916) <= |b*b|) &&
  @      (a*c==0 || 2^^(-916) <= |a*c|) &&
  @      |b| <= 2^^510 &&
  @      |a| <= 2^^995 &&
  @      |c| <= 2^^995 &&
  @      |a*c| <= 2^^1020
  @ ensures \result==0 ||
  @     |\result-(b*b-a*c)| <= 2*ulp(\result)
  @ */

double Kahan_discr
       (double a, double b, double c) {
  double p,q,d,dp,dq;
  p=b*b;
  q=a*c;

  if (p+q <= 3*fabs(p-q))
    d=p-q;
  else {
    dp=exactmult(b,b,p);
    dq=exactmult(a,c,q);
    d=(p-q)+(dp-dq);
  }
  return d;
}
```

8 About the ulp

The previous result gives an error bound expressed in a number of ulps of the floating-point result. It is usually a better idea to give the results in terms of a number of ulps of the exact real result [14]. In the original article [7], there is no clear error bound. "Accurate" is a fuzzy enough word to encompass error bounds in terms of $\mathrm{ulp}(d)$ or in terms of $\mathrm{ulp}(b^2 - ac)$.

The problem is that the same property does not hold when replacing $\mathrm{ulp}(d)$ by $\mathrm{ulp}(b^2 - ac)$, whatever definition is used for the ulp of a real value. As a counterexample, let

$$\begin{aligned} a &= 1.00010110111001110110010111 \\ &\quad 00000000000000000000000000 \times 2^{20}, \\ b &= 1.01101010000010011110011001 \\ &\quad 10011111110011101111001101 \times 2^{26}, \\ c &= 1.11010101111101000011100011 \\ &\quad 11001000000000000000000000 \times 2^{31}. \end{aligned}$$

Then, $q = (2^{52})^+$, and $p = (2^{53})^+ = 2q$. We are in the first case of the test, and $d = q > 2^{52}$. So

$$\begin{aligned} d &= 1.00000000000000000000000000 \\ &\quad 00000000000000000000000001 \times 2^{52}, \\ b^2 - ac &= 1.11111111111111111111111111 \\ &\quad 11111111111111111111111101111[\ldots] \times 2^{51}. \end{aligned}$$

So $b^2 - ac < 2^{52} < d$. Therefore, the ulps differ: $\mathrm{ulp}(d) = 1$, but then, $\mathrm{ulp}(b^2 - ac) = 0.5$ (this is true for all the definitions of the real ulp in the literature [14]), so d and $b^2 - ac$ do not share the same ulp. Now, we compute $\delta = |d - (b^2 - ac)| > 1.25$. On one hand, we have $\delta \leq 2\mathrm{ulp}(d)$ (as proved). On the other hand, we have

$$|d - (b^2 - ac)| > 2\mathrm{ulp}(b^2 - ac).$$

This is extremely surprising and unexpected as d and $b^2 - ac$ are very near one another. This also means that the error bound is really tight. Fortunately, the author did not find any results in the literature based on the claimed property with the incorrect ulp. Note that this was incorrectly stated in [19]: the ulps stated there also were the floating-point ulps and not the ulps of the real exact value.

The inequality $|d - (b^2 - ac)| \leq 4 \times \mathrm{ulp}(b^2 - ac)$ is true anyway. The floating-point and the exact result cannot be very far away. However, this last statement has multiplied the previous error bound by 2 everywhere except just under the power of the radix.

9 Conclusion

All the formal proofs were done "by hand." The conclusion of that is that part of them must be automated. The proofs on overflow by hand were really cumbersome, and therefore, the bounds are not very tight on that point of view. The bounds sufficient to guarantee that there is no overflow could probably be multiplied by 2 and still hold. But it would have been too much a bore. This automation is currently in progress using the Gappa tool [21]. This should give better bounds on overflow that would also be certified by Coq proofs.

The program proved is limited to IEEE-754 double precision with rounding to nearest, ties to even. Analogous facts could be

proved with other reasonable formats: we need a precision greater than 4 and underflow and overflow thresholds such that $2^{\pm 3prec}$ are normal floating-point numbers. This of course includes all IEEE-754 formats. Regarding rounding, rounding to nearest is needed, but the tie may be any choice, as long as it is consistent (it must only depend on the value and not on the state of the processor). So this result also holds for rounding to nearest, ties away from zero, of the 754R.

This case study has validated our method in several ways:

- The annotations are short and understandable by people who have never met a formal proof checker.
- Some preceding results on Veltkamp/Dekker algorithms have been reused without any difficulty.
- A known difficult algorithm has been proved.
- An unexpected difficulty that does not appear in the initial proof (the fact that the decisive test `p+q<=3*fabs(p-q)` can be wrong) has been isolated and solved.
- Precise sufficient conditions, including underflow and overflow hypotheses, have been given and proved.

This work unfortunately agrees with the initial article on the fact that the proofs are still "far longer and trickier than the algorithms and programs in question" [7]. The annotations are just shorter than the program, but the pen-and-paper proof is noticeably long for such a short program, and the Coq proof is nearly 5,000 lines long (plus about 7,000 for the Veltkamp/Dekker algorithm). The ratio 1 line of C for 500 lines of (formal) proof will certainly not convince Kahan. Nevertheless, this work may fill the section "Proofs—TO BE APPENDED SOME DAY" of [7].

ACKNOWLEDGMENTS

The author would like to thank M. Daumas and G. Melquiond for their initial work on the subject and constructive discussions. The author is also deeply indebted to the reviewers for their perspicacity on incorrect or vague statements. This research was supported by the ANR-05-BLAN-0281-04 CerPAN project.

REFERENCES

[1] T.J. Dekker, "A Floating Point Technique for Extending the Available Precision," *Numerische Mathematik,* vol. 18, no. 3, pp. 224-242, 1971.
[2] S. Boldo and M. Daumas, "A Mechanically Validated Technique for Extending the Available Precision," *Proc. 35th Asilomar Conf. Signals, Systems, and Computers,* pp. 1299-1303, http://perso.ens-lyon.fr/marc.daumas/SoftArith/BolDau01b.pdf, 2001.
[3] F. de Dinechin, C.Q. Lauter, and J.-M. Muller, "Fast and Correctly Rounded Logarithms in Double-Precision," *Theoretical Informatics and Applications,* vol. 41, pp. 85-102, http://perso.ens-lyon.fr/florent.de.dinechin/recherche/publis/2007-TIA.pdf, 2007.
[4] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding," *ACM Trans. Math. Software,* vol. 33, no. 2, p. 13, http://doi.acm.org/10.1145/1236463.1236468, June 2007.
[5] S. Boldo and M. Daumas, "A Simple Test Qualifying the Accuracy of Horner's Rule for Polynomials," *Numerical Algorithms,* vol. 37, nos. 1-4, pp. 45-60, http://www.ingentaconnect.com/content/klu/numa/2004/00000037/F0040001/05276602, 2004.
[6] N.J. Higham, *Accuracy and Stability of Numerical Algorithms,* SIAM, second ed., http://www.ma.man.ac.uk/~higham/asna/, 2002.
[7] W. Kahan, *On the Cost of Floating-Point Computation without Extra-Precise Arithmetic,* p. 21, http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf, Nov. 2004.
[8] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C.K. Yap, "Classroom Examples of Robustness Problems in Geometric Computations," *Proc. 12th European Symp. Algorithms (ESA '04),* pp. 702-713, http://cgal.inria.fr/Publications/2004/KMPSY04, 2004.
[9] M. Daumas, L. Rideau, and L. Théry, "A Generic Library of Floating-Point Numbers and Its Application to Exact Computing," *Proc. 14th Int'l Conf. Theorem Proving in Higher Order Logics (TPHOLs '01),* pp. 169-184, http://perso.ens-lyon.fr/marc.daumas/SoftArith/DauRidThe01.ps, 2001.
[10] S. Boldo, "Preuves Formelles en Arithmétiques à Virgule Flottante," PhD dissertation, École Normale Supérieure de Lyon, http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2004/PhD2004-05.pdf, Nov. 2004.
[11] J.-C. Filliâtre and C. Marché, "Multi-Prover Verification of C Programs," *Proc. Sixth Int'l Conf. Formal Eng. Methods (ICFEM '04),* pp. 15-29, http://www.lri.fr/~filliatr/ftp/publis/caduceus.ps.gz, Nov. 2004.
[12] J.-C. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus Platform for Deductive Program Verification," *Proc. 19th Int'l Conf. Computer Aided Verification (CAV '07),* W. Damm and H. Hermanns, eds., http://www.lri.fr/~filliatr/ftp/publis/cav07.pdf, July 2007.
[13] S. Boldo and J.-C. Filliâtre, "Formal Verification of Floating-Point Programs," *Proc. 18th IEEE Symp. Computer Arithmetic (ARITH '07),* P. Kornerup and J.-M. Muller, eds., pp. 187-194, June 2007.
[14] J.-M. Muller, *On the Definition of ulp(x),* Technical Report LIP RR2005-09 INRIA RR-5504, Laboratoire de l'Informatique du Parallélisme, ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RR/RR-5504.pdf, 2005.
[15] G.W. Veltkamp, *Algolprocedures voor het Berekenen van een Inwendig Product in Dubbele Precisie,* RC-Informatie 22, Technische Hogeschool Eindhoven, 1968.
[16] G.W. Veltkamp, *ALGOL Procedures voor het Rekenen in Dubbele Lengte,* RC-Informatie 21, Technische Hogeschool Eindhoven, 1969.
[17] A.H. Karp and P. Markstein, "High-Precision Division and Square Root," *ACM Trans. Math. Software,* vol. 23, no. 4, pp. 561-589, Dec. 1997.
[18] S. Boldo, "Pitfalls of a Full Floating-Point Proof: Example on the Formal Proof of the Veltkamp/Dekker Algorithms," *Proc. Third Int'l Joint Conf. Automated Reasoning (IJCAR '06),* pp. 52-66, Aug. 2006.
[19] S. Boldo, M. Daumas, W. Kahan, and G. Melquiond, "Proof and Certification for an Accurate Discriminant," *Proc. 12th IMACS-GAMM Int'l Symp. Scientific Computing, Computer Arithmetic and Validated Numerics (SCAN '06),* Sept. 2006.
[20] P.H. Sterbenz, *Floating Point Computation.* Prentice Hall, 1974.
[21] G. Melquiond and S. Pion, "Formally Certified Floating-Point Filters for Homogeneous Geometric Predicates," *Theoretical Informatics and Applications,* vol. 41, no. 1, pp. 57-70, http://perso.ens-lyon.fr/guillaume.melquiond/doc/07-tia.pdf, 2007.