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

Manuscript received July 22, 2007.

S. Boldo is with the INRIA Saclay Île de France.

Abstract

This article tackles Kahan's algorithm to compute accurately the discriminant whatever the inputs. This is a known difficult problem and this algorithm leads to an error bounded by 2 ulps of the result. The proofs involved are long and tricky and even trickier than expected as the test involved may give an unexpected result. We give here the total demonstration of the validity of this algorithm and we provide sufficient conditions to guarantee neither Overflow, nor Underflow will jeopardize the result. The program was annotated using the Caduceus tool and the proof obligations were done using the Coq automatic proof checker.

Index Terms

Floating-point, discriminant, formal proof.

I. Introduction

Floating-point arithmetic is a field that seems dangerous and obscure to most programmers. However, floating-point experts can create very tricky algorithms that take advantage of inexact computation to get correct or accurate results.

This category includes for example:

- expansions 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],
- 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 Professor 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, Overflow and Underflow included. We may have too tight bounds, 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 page 4) computes the value of the discriminant, meaning given a, b and c, it computes $b^2 - ac$. This is of course useful to compute the zeroes of the polynomial $ax^2+2bx+c$, but also for computing the orientation test. 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.

For example, 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 the incorrect answer due to round-off errors [8].

The responsibility of this 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 n and e bounded and has a value equal to $n \times 2^e$. This formalization has been able to prove both old and new results [10].

As we want to prove the real program, we use 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 on $prec$ bits (in IEEE double precision, $prec = 53$). μ is the smallest (subnormal) floating-point number (in IEEE double precision, $\mu = 2^{-1074}$).

The unit in the last place is denoted by ulp. The successor of a float x is denoted as x^+ and its predecessor 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 greater or equal to the minimal exponent. The real value associated to x is then $n_x \times 2^{e_x}$.

II. 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, even, if the programmer does not change it). We then rewrite the initial program in the C language (Program 1).

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

- `fabs`, that computes the absolute value of a floating-point number,
- `exactmult`, that 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. This algorithm was discovered at the same period by Veltkamp and Dekker [1], [14], [15] 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 [16].

The function `fabs` is assumed to give the correct answer (that is to say the float which value is the absolute value of the input). The function `exactmult` is proved to be correct. The proof of this algorithm is described in [17] whatever the radix. The program given here does not rely on any other complex function: the demonstration is self-contained.

III. Proof of the "ideal" algorithm

This partial proof has already been published in [18]. The main ideas of the proof are nevertheless explained, as there was not space enough in [18] and as part of them will be used afterwards. The notations are those of Program 1.

The "ideal" algorithm means that we assume the test is made on real values: it is not `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.

A. Proof

I assume for this section there is neither Underflow, nor Overflow. Let us prove that d is within 2 ulps of $b^2 - ac$. Let $\delta = |d - (b^2 - ac)|$.

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

1) First case: $3|p-q| \geq p+q$*:* This corresponds to the first possibility in the `if`. We here split into two subcases depending on the respective values of p and q.

a) 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 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 q is negative.

  Then p and $-q$ share the same sign and $d \geq p$ and $d \geq |q|$ so we easily have that $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(p) + \frac{1}{2}\mathrm{ulp}(q) \leq \frac{3}{2}\mathrm{ulp}(d)$.

b) Second subcase: $q \geq p$*:* Then $q \geq 0$ and $|p-q| = -p+q$ 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 $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(d) + \mathrm{ulp}(d) = 2\mathrm{ulp}(d)$.

2) Second case: $3|p-q| < p+q$*:* This corresponds to the second possibility in the `if`. This case is much more complex. We prove several intermediate results before splitting into cases.

First, if $p = q$ then d is correct within half an ulp. Let us now assume that $p \neq q$.

a) The computation of $p - q$ *is correct:*

If $q \leq 0$, then we have $3|p-q| < p+q = |p| - |q| \leq |p-q|$ that is absurd. Therefore $q > 0$. So $p = |p| \leq |p-q| + |q| \leq \frac{1}{3}(p+q) + q$ 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 [19].

b) Bounding $\circ(dp - dq)$*:*

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

So $|dp - dq| \leq |dp| + |dq| \leq \frac{1}{2}\mathrm{ulp}(p) + \frac{1}{2}\mathrm{ulp}(q) \leq \frac{3}{2}\min(\mathrm{ulp}(p), \mathrm{ulp}(q)) \leq \frac{3}{2}|p-q|$. 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) is not far from the lowest possible bound (just under 1.5).

c) Evaluation of δ*:*

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

The first easy case is when $\mathrm{ulp}(p) = \mathrm{ulp}(q)$, then the computation of $dp - dq$ is exact and $\delta \leq \frac{1}{2}\mathrm{ulp}(d)$. We now split into two subcases.

First, we assume we are in the general case corresponding to $|p-q| \geq 3\min(\mathrm{ulp}(p), \mathrm{ulp}(q))$.

Then $|dp - dq| \leq \frac{3}{2}\min(\mathrm{ulp}(p), \mathrm{ulp}(q)) \leq \frac{1}{2}|p-q|$. So $|p - q + \circ(dp - dq)| \geq \frac{1}{2}|p-q| \geq |dp - dq|$. In that case, we have $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(\circ(dp - dq)) \leq \mathrm{ulp}(d)$.

Now, we assume 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.

We first assume that $p \geq q$ as p and q are here symmetrical. We then shift the exponents of

p and q, so that they are near 1. The cases are then either ($p = 1$, $q = 1^-$), or ($p = 1$, $q = 1^{--}$) or ($p = 1^+$, $q = 1^-$). Note that the latest case fits into the general case.

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

- Assume dp and dq share the same sign.

  Then we prove that the computation of $dp - dq$ is exact that implies that $\delta \leq \frac{1}{2}\mathrm{ulp}(d)$.

  If $dq = 0$ then $dp - dq$ is exactly dp. If not, then dp fits on $prec + 1$ bits with exponent $-2prec$ and dq fits on $prec$ bits with exponent $-2prec$. So $dp - dq$ can use the exponent $-2prec$. More, as $|dp| \leq 2^{-prec}$, $|dp - dq| < 2^{-prec}$ so $|dp - dq| \leq 2^{-prec} - 2^{-2prec}$ that fits into $prec$ bits. So $dp - dq$ is computed exactly.
- Let us now assume that dp and dq have opposite signs. We split one more time into two subcases.

  - Assume $dp - dq \geq 0$.

    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 $\mathrm{ulp}(\circ(dp - dq)) \leq 2^{1-2prec}$.

    Finally, we bound δ by $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + \frac{1}{2}\mathrm{ulp}(\circ(dp - dq))$.

    Then $\delta \leq \frac{1}{2}\mathrm{ulp}(d) + 2^{-2prec} \leq \frac{1}{2}\mathrm{ulp}(d) + 2^{-prec}d$ and $\delta \leq \frac{3}{2}\mathrm{ulp}(d)$.
  - Assume $dp - dq \leq 0$.

    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 $\delta \leq \frac{1}{2}\mathrm{ulp}(d)$.

    As dp and dq accept $-2prec$ as exponent, and $|dp - dq| \leq 2^{-1-prec} + 2^{-1-prec} = 2^{-prec}$, we either have $|dp| = |dq| = 2^{-1-prec}$ where $dp - dq = -2^{-prec}$ is representable; or we have $|dp - dq| < 2^{-prec}$ so it fits on $prec$ bits with exponent $-2prec$.

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

IV. Proof of the program

We now prove that the real-life program satisfies the same property, meaning that d is within 2 ulps of $b^2 - ac$. 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 5-digit mantissa. Let $p = 27 = 11011_2$ and $q = 14 = 1110_2$, then $p+q = 41 > 3|p-q| = 3 \times 13 = 39$. But $\circ(p+q) = \circ(41) = 40$ and $\circ(p-q) = p-q = 13$ and $\circ(3 \circ (p-q)) = \circ(39) = 40$. Therefore

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

so the floating-point and the real test disagree.

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

A. 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 $p \geq q$.

We prove some intermediate results and then cut down and solve all subcases.

1) Neighboring of p and q:

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

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

$p+q-3(p-q) \leq \text{ulp}(\circ(3|p-q|)) \leq 2^{1-prec} \circ (3|p-q|) \leq 2^{1-prec} \times 3(p-q)\frac{1}{1-2^{-prec}}$. So $-2p+4q \leq (3p-3q)\frac{2^{1-prec}}{1-2^{-prec}}$ and therefore $q \leq p\frac{1+2^{1-prec}}{2+2^{-prec}}$.

2) Exponents of d and q:

If $e_d \geq e_q$, then $\delta = |p-q-(b^2-ac)| \leq \frac{1}{2}(\text{ulp}(p)+\text{ulp}(q)) \leq \frac{3}{2}\text{ulp}(q) \leq \frac{3}{2}\text{ulp}(d)$. Let us now assume that $e_d < e_q$.

3) Same exponents for p and q:

As p and q are positive, this implies that either: $\text{ulp}(p) = \text{ulp}(q)$ or $\text{ulp}(p) > \text{ulp}(q)$.

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

4) Different exponents for p and q:

If $\text{ulp}(p) > \text{ulp}(q)$, we know that $p \leq 2q \leq p\frac{1+2^{1-prec}}{1+2^{-1-prec}}$. We want to prove that $2q = p^+$.

First, $p = 2q$ is impossible as $3|p-q| < p+q$, so $p < 2q$ and $p^+ \leq 2q$.

Next, $p^{++} \geq p + 2\text{ulp}(p) \geq p + \frac{2p}{2^{prec}-1} = p\frac{2^{prec}+1}{2^{prec}-1}$. So $p^{++} > p\frac{1+2^{1-prec}}{1+2^{-1-prec}} \geq 2q$ so $p^+ \geq 2q$.

This case is in fact impossible. This is achieved by looking at all the possible cases for n_q:

- If q is a power of 2, then $q = 2^{prec-1+e_q}$ and then $p^+ = 2^{prec-1+(1+e_q)}$ and $p = (2^{prec}-1)2^{e_q}$ so $\mathrm{ulp}(q) = \mathrm{ulp}(p)$ which is absurd.
- 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 $2^{prec-1+e_q} > d = p - q = q - 2\mathrm{ulp}(q) = (n_q - 2)2^{e_q}$ so $n_q < 2^{prec-1} + 2$. This implies 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.

B. 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 $p \geq q$.

1) First facts:

We have $p-q \leq \circ(p-q)\,(1+2^{-prec}) \leq \frac{1}{3}\circ(3\circ(|p-q|))\,(1+2^{-prec})^2 \leq \frac{1}{3}\circ(p+q)\,(1+2^{-prec})^2$, so $p - q \leq \frac{1}{3}(p+|q|)\frac{\left(1+2^{-prec}\right)^2}{1-2^{-prec}}$.

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

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

2) Bounding $dp - dq$:

Next, $p - q \leq \frac{1}{3}(p+q)\frac{\left(1+2^{-prec}\right)^2}{1-2^{-prec}}$ implies that $p \leq 3q$.

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

3) Bounding δ:

Then $\circ(p-q) + \circ(dp-dq) \geq \circ(p-q) - 3\mathrm{ulp}(\circ(p-q))$ so $d \geq \circ(p-q) - 3\mathrm{ulp}(\circ(p-q))$ and $\mathrm{ulp}(d) \geq \frac{1}{2}\mathrm{ulp}(\circ(p-q))$.

Now, we can end the proof:

$\delta \leq \frac{1}{2}(\mathrm{ulp}(d)+\mathrm{ulp}(\circ(p-q))+\mathrm{ulp}(\circ(|dp-dq|))) \leq \mathrm{ulp}(d)\left(\frac{1}{2} + 1 + 2^{1-prec} \times 3 \times 2\right) \leq 2\mathrm{ulp}(d)$.

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

V. Overflow

Overflow is usually disregarded as it usually creates non-numerical quantities (NaN or $\pm\infty$). Unfortunately, it may happen that Overflow occur 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 or equal to the maximal floating-point number in IEEE double precision that is to say $(2-2^{-52})\,2^{1023}$. This is done using a command line option of Caduceus 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 NaNs.

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

We then require $|a|$ and $|c|$ to be smaller than 2^{995} so that $(2^{27}+1)\times a$ or c does not overflow. We also require $|a \times c|$ to be smaller than 2^{1020} and $|b| \leq 2^{510}$ so that the other computations, including especially $3 \times \circ(p-q)$ do not overflow.

VI. 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 afterwards. 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 afterwards to improve it is a huge benefit.

The proved sufficient conditions for the preceding proof are:

- p, q and d are either zeroes 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 intuitive used theorems 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 zeroes 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 a intermediary value. We therefore give sufficient conditions to guarantee these requirements. We need either b to be 0 or b^2 to be greater than $2^{3prec-1}\mu$ and we need either $a \times c$ to be 0 or $|a \times c|$ to be greater than $2^{3prec-1}$.

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.

VII. Fully annotated program

The fully annotated program is Program 2. The annotations interpreted by Caduceus are comments beginning with `@`. Details on the syntax of Caduceus' annotations can be found in [11], [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.

VIII. Conclusion

All the formal proofs were done "by hand". The conclusion of that is that part of them must be automatized. 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 there is no

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);
}
```

Program 2 Fully annotated accurate discriminant computation (continued)

```
/*@ 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;
}
```

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 [20]. This should give better bounds on Overflow that would also be certified by Coq proofs.

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 that Veltkamp/Dekker algorithms have been re-used 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.

The main drawback of this work is does not fulfill the observation 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 five thousand lines long (plus about seven thousand for the Veltkamp/Dekker algorithm). The ratio 1 line of C for 500 lines of (formal) proof will certainly not convince Prof. Kahan.

ACKNOWLEDGEMENTS

The author would like to thank M. Daumas and G. Melquiond for their initial work on the subject and constructive discussions.

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," in *35th Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, 2001, pp. 1299–1303. [Online]. Available: http://perso.ens-lyon.fr/marc.daumas/SoftArith/BolDau01b.pdf

[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, 2007. [Online]. Available: http://perso.ens-lyon.fr/florent.de.dinechin/recherche/publis/2007-TIA.pdf

[4] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, p. 13, June 2007, article 13, 15 pages. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236468

[5] S. Boldo and M. Daumas, "A simple test qualifying the accuracy of horner's rule for polynomials," *Numerical Algorithms*, vol. 37, no. 1-4, pp. 45–60, 2004. [Online]. Available: http://www.ingentaconnect.com/content/klu/numa/2004/00000037/F0040001/05276602

[6] N. J. Higham, *Accuracy and stability of numerical algorithms*. SIAM, 2002, second Edition. [Online]. Available: http://www.ma.man.ac.uk/~higham/asna/

[7] W. Kahan, "On the cost of floating-point computation without extra-precise arithmetic," World-Wide Web document, p. 21, Nov. 2004. [Online]. Available: http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf

[8] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. K. Yap, "Classroom examples of robustness problems in geometric computations," in *Proc. 12th European Symposium on Algorithms*, ser. Lecture Notes Comput. Sci., vol. 3221. Springer-Verlag, 2004, pp. 702–713. [Online]. Available: http://cgal.inria.fr/Publications/2004/KMPSY04

[9] M. Daumas, L. Rideau, and L. Théry, "A generic library of floating-point numbers and its application to exact computing," in *14th International Conference on Theorem Proving in Higher Order Logics*, Edinburgh, Scotland, 2001, pp. 169–184. [Online]. Available: http://perso.ens-lyon.fr/marc.daumas/SoftArith/DauRidThe01.ps

[10] S. Boldo, "Preuves formelles en arithmtiques virgule flottante," Ph.D. dissertation, École Normale Suprieure de Lyon, Nov. 2004. [Online]. Available: http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2004/PhD2004-05.pdf

[11] J.-C. Filliâtre and C. Marché, "Multi-Prover Verification of C Programs," in *Sixth International Conference on Formal Engineering Methods (ICFEM)*, ser. Lecture Notes in Computer Science, vol. 3308. Seattle: Springer-Verlag, Nov. 2004, pp. 15–29. [Online]. Available: http://www.lri.fr/~filliatr/ftp/publis/caduceus.ps.gz

[12] Jean-Christophe Filliâtre, Claude Marché and Thierry Hubert, "The Caduceus tool for the verification of C programs," http://caduceus.lri.fr/.

[13] S. Boldo and J.-C. Filliâtre, "Formal verification of floating-point programs," in *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, P. Kornerup and J.-M. Muller, Eds., Montpellier, France, June 2007, pp. 187–194.

[14] G. W. Veltkamp, "Algolprocedures voor het berekenen van een inwendig product in dubbele precisie," Technische Hogeschool Eindhoven, RC-Informatie 22, 1968.

[15] ——, "ALGOL procedures voor het rekenen in dubbele lengte," Technische Hogeschool Eindhoven, RC-Informatie 21, 1969.

[16] A. H. Karp and P. Markstein, "High-precision division and square root," *ACM Transactions on Mathematical Software*, vol. 23, no. 4, pp. 561–589, Dec. 1997.

[17] S. Boldo, "Pitfalls of a full floating-point proof: Example on the formal proof of the veltkamp/dekker algorithms," in *Proceedings of the third International Joint Conference on Automa ted Reasoning (IJCAR)*, Seattle, USA, aug 2006, pp. 52–66.

[18] S. Boldo, M. Daumas, W. Kahan, and G. Melquiond, "Proof and certification for an accurate discriminant," in *12th IMACS-GAMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, Duisburg,Germany, sep 2006.

[19] P. H. Sterbenz, *Floating point computation*. Prentice Hall, 1974.

[20] 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, 2007. [Online]. Available: http://perso.ens-lyon.fr/guillaume.

melquiond/doc/07-tia.pdf