

Twin-Float Arithmetic

John Abbott

Abstract

We present a heuristically certified form of floating-point arithmetic. As prerequisite we need floating-point arithmetic where the user can set the precision. Our work is developed from the idea of paired floats expounded by Traverso and Zanoni [2]. Twin-float arithmetic is suitable for use only where the input data are exact (or can be obtained to high enough precision). The arithmetic includes a (heuristic) zero test, and so can be used in Buchberger’s Algorithm. The ideas presented here are implemented as a `ring` in CoCoALib, called `RingFloat`, allowing them to be used in a wide variety of algebraic computations including Gröbner bases.

1 Introduction and Motivation

The principal aim of the arithmetic described in this paper is to obtain quickly a good approximation to the exact result of a computation on exact input data (*i.e.* with rational coefficients). This aim contrasts with that of several recent works which endeavour to tackle problems whose input data have approximate coefficients (*e.g.* GCD of approximate polynomials). The techniques described in this article are not applicable to problems whose input is approximate.

The work presented here is a development of ideas proposed by Traverso and Zanoni [2]. Their original idea was to find a way of computing a good approximation to a Gröbner basis over the rationals but without the prohibitive cost of the rational arithmetic. They could not make direct use of floating point numbers because the zero test, essential to Buchberger’s Algorithm, is unreliable between floats. They experimented with a “hybrid” of floating point and modular arithmetic: the zero test became reliable but the final coefficient values were typically swamped by rounding error. Finally, they invented the idea of using a pair of floating point values at different precisions: this permitted a heuristically reliable zero test, and also allowed the growth of rounding error to be monitored. This last idea did indeed make it possible to compute Gröbner bases rapidly: the structure is determined correctly, and the coefficients are heuristically guaranteed approximations to their true rational values.

Twin-float arithmetic, implemented in CoCoALib, is a further development of Traverso and Zanoni’s paired floats, abstracted from the context of Buchberger’s Algorithm. During the process of abstraction we resolved some outstanding issues which were irrelevant in the original context. A recent addition to the arithmetic is the ability to recover a rational number from a twin-float value. In some cases this capability allows the recovery of the exact Gröbner basis from one computed using twin-floats.

1.1 An Intuitive Description of Twin-Floats

Twin-float arithmetic presupposes the availability of normal floating point arithmetic where the user may choose the precision. For instance, in CoCoALib the implementation is based upon the `mpf_t` type offered by the GMP library [5].

The fundamental idea is that each twin-float value comprises a floating point value together with a good heuristic estimate of its accuracy (*i.e.* the number of reliable significant bits); the only special case is zero, which is always regarded as being exact. Before computation begins, the user specifies a minimum acceptable accuracy, or equivalently, a maximum permissible relative error for the values. Every arithmetic operation on twin-float values checks that the estimated accuracy is sufficient; if not, the operation fails (*e.g.* in the CoCoALib implementation an exception is thrown). However, an addition or subtraction which results in enough cancellation is regarded as producing exactly zero.

In general, if a computation on twin-floats does not fail then it produces a result with (at least) the minimum requested accuracy. However, in very rare cases the cancellation heuristic can lead to wrong results: for instance, at low precision we could obtain that $(114243/80782)^2 = 2$.

1.2 Comparison with Interval Arithmetic

A natural question is: *how does twin-float arithmetic compare with interval arithmetic?* One important difference is that values in interval arithmetic are always guaranteed correct (*i.e.* the interval surely contains all possible results), whereas twin-float values are only heuristically likely to be correct to the requested accuracy (or the computation could even fail).

A direct consequence of the strong guarantee in interval arithmetic is that the intervals can, and generally do, become excessively large. For example, if X is an interval of width ε then the computation $X - X$ will yield an interval of width 2ε centred on 0. In contrast, thanks to the cancellation heuristic, twin-float arithmetic will always yield the exact value zero whenever we evaluate $X - X$.

A more subtle case where a needlessly wide interval is produced happens when computing $Y \cdot (1 - Y)$ with $Y = [0, 1]$; we obtain the interval $[0, 1]$ rather than the best interval $[0, 1/4]$. Overly wide intervals tend to be produced when the two operands to an arithmetic operation are not independent.

Another difference important to us is that interval arithmetic does not naturally include a zero test offering strong guarantees of correctness. Of course, a heuristic zero test could be devised, but then the advantages over twin-floats would largely disappear.

Finally we note that interval arithmetic can be used in cases where input coefficients are approximate whereas twin-floats cannot. Since we have explicitly excluded approximate inputs, this extra capability is of no benefit in our context.

2 Definition of Twin-Float Arithmetic

The overall intention of the design was to produce a useful compromise between the speed of finite precision floating-point computations and the unswerving accuracy of rational arithmetic. Achieving both characteristics simultaneously for all computations appears to be impossible. Our compromise is to accept a design which achieves both “most of the time”, but for which there is a small chance that we fail to obtain a result and an even smaller chance that we obtain a wrong result. The idea is to use heuristics to verify the accuracy of the result of each individual operation.

Before any computation commences a global minimum accuracy (in bits) must be specified; we shall call this value B , it is a positive integer (*e.g.* 256). We use heuristics to estimate the accuracy of the result of each twin-float operation. If the heuristic accuracy is below the minimum requested (*e.g.* after subtracting two close values) then **failure occurs**; in our CoCoALib implementation an `InsufficientPrecision` exception is thrown. We observe that any arithmetic operation (including comparison) could fail in this way.

Since twin-float arithmetic is based upon finite precision floating point representations, phenomena of overflow and underflow can occur. We delegate the responsibility for reporting the two types of failure to the underlying floating point implementation. In CoCoALib we use the data type `mpf_t` of the GMP library [5] which offers a range wide enough for almost all purposes, so problems of overflow or underflow are highly unlikely in normal computations.

2.1 Representation of a Twin-Float Value

We recall that prior to any computation with twin-float values a global minimum accuracy is specified: B bits. This value remains constant throughout the computation. Additionally, we use two more global constants, both are positive integer bit counts: N is the initial noise level (see 2.4), and S is width of the “safety zone”.

A twin float value is represented as a pair of floating point components whose values are almost equal, (V_1, V_2) . Zero has a special representation, $(0, 0)$. All other values are represented by two non-zero components. For the representation to be valid the first B bits of V_1 and V_2 must agree, equivalently their relative difference must have magnitude at most 2^{-B} . We shall see later (see section 2.5.2) that special action is taken to avoid the relative difference becoming too small.

Naturally both components must be floating point values with mantissas at least B bits long. Indeed, the rule for creating twin-float values from rational numbers (see section 2.4) implies that the mantissas must each have at least $B + S + N$ bits of precision.

2.1.1 An Alternative Representation

We mention here for completeness a more compact representation which offers some slight practical benefits. In the rest of this article we shall assume use of the main representation as it permits a clearer presentation. The simple exercise of adapting the rest of the article to this alternative representation is left to the interested reader.

Instead of representing each twin-float two as almost equal values (V_1, V_2) , we use the pair (V_1, δ) where $\delta = (V_2 - V_1)/V_1$ is the signed relative difference. The main advantage is that the relative difference can be held as a low precision value thus saving space and perhaps also computation time — at best this reduces memory space by a factor of two, and increases speed by a factor of two compared to the main representation.

2.2 The Natural Meaning of Twin Floats

Unsurprisingly the definition of equality between twin-float values is closely related to the meaning we give to the representation. In this subsection we describe the most natural way of interpreting the meaning of a twin-float value, and then present the resulting equality test. We then show that the equality test has some potentially undesirable properties, *viz.* a “drastic” failure of transitivity. In the next subsection we describe an alternative, more conservative interpretation of a twin-float whose corresponding equality test is better behaved.

The special value $(0, 0)$ is interpreted as being **exactly zero**. Otherwise let $V = (V_1, V_2)$ be a non-zero twin-float value, and define $\delta = (V_2 - V_1)/V_1$ to be the signed relative difference. Define $\varepsilon_o = \min\{2^k : k \in \mathbb{Z} \text{ and } 2^k > |V_1 - V_2|\}$, the least power of two exceeding the absolute difference.

We define two intervals related to V . The **outer interval** is $\text{outer}(V) = [V_1 - \varepsilon_o, V_1 + \varepsilon_o]$, and the **inner interval** is $\text{inner}(V) = [V_1 - \varepsilon_i, V_1 + \varepsilon_i]$ where $\varepsilon_i = \varepsilon_o/2^{N/2}$. We regard the value represented by V as lying “very probably” in the inner interval, and “absolutely certainly” in the outer interval.

2.2.1 The Equality Test for the Natural Interpretation

We recall that the equality test is permitted to produce three mutually exclusive outcomes: *true*, *false*, and *failure*. The outcome *failure* indicates that it is not possible to determine whether the two values are equal or unequal with adequate certainty. The equality test is heuristic and optimistic: if two values are “very probably equal” then the test says they are actually equal.

Let $V = (V_1, V_2)$ and $W = (W_1, W_2)$ be two twin-floats. If they are both zero then they are surely equal. If one is zero and the other not then they are surely unequal. Otherwise we have the general case where both values are non-zero. If $\text{outer}(V) \cap \text{outer}(W) = \emptyset$ then the values are unequal. If $\text{inner}(V) \cap \text{inner}(W) \neq \emptyset$ then the values are equal. Otherwise we cannot say, and return *failure*.

With this definition of equality we can get a “drastic” failure of transitivity: *i.e.* there exist values X, Y , and Z such that $X = Y$ and $Y = Z$ both yield *true* while $X = Z$ yields *false*. This can happen if the interval $\text{inner}(Y)$ is particularly wide, namely if $|\text{inner}(Y)| > \frac{1}{2}(|\text{outer}(X)| + |\text{outer}(Z)|)$.

2.3 A Conservative Meaning for Twin-Floats

Define the **mantissa**, $\text{mant}(V_1)$, and **exponent**, $e(V_1)$, of a non-zero floating point value V_1 as follows: $e(V_1)$ is that integer and $\text{mant}(V_1)$ is that real number with $1 \leq |\text{mant}(V_1)| < 2$ for which $V_1 = \text{mant}(V_1) \times 2^{e(V_1)}$.

Given a non-zero twin-float $V = (V_1, V_2)$, let W be the best B -bit approximation to V_1 , that is $W = m_w \times 2^{e(V_1) - B + 1}$ where m_w is the nearest integer to $2^{B-1} \times \text{mant}(V_1)$, and in the case of a tie we round away from zero.

Now we define the interval $\text{outer}^*(V) = [W - \varepsilon_o^*, W + \varepsilon_o^*]$ where $\varepsilon_o^* = 2^{e(W) - B + 1}$. So $\text{outer}^*(V)$ comprises exactly those values which agree with V_1 to at least B bits. We shall also require that any valid twin-float V satisfies $|\text{inner}(V)| < \frac{1}{2}|\text{outer}^*(V)|$.

2.3.1 The Equality Test for the Conservative Interpretation

The test is the same as for the natural interpretation except that we use outer^* intervals instead of outer intervals.

With the given upper limit on the width of inner intervals we are safe from “drastic” failures of transitivity, though “mild” failures may still occur: *i.e.* there exist twin-floats X, Y , and Z such that $X = Y$ and $Y = Z$ both give *true* while $X = Z$ produces *failure*. We believe this behaviour is preferable to a “drastic” failure.

There is a price to pay with this conservative approach: with the natural interpretation, the outer interval gives a good estimate of the true accuracy of the value; whereas here we use the potentially far wider outer^* interval, probably ignoring much of the true accuracy. As a consequence, some equality tests may needlessly produce *failure* despite having enough information to be certain that the values were unequal.

2.4 Conversion of Rational Numbers into Twin-Floats

The first step in computing with twin-floats is to convert the exact input data (*i.e.* rational numbers) into twin-float representation. And to convert a rational number n/d , we simply convert the integers n and d independently, and then compute the quotient of their images. So it remains to describe how to convert an integer.

We are given an integer n , and wish to produce its twin-float equivalent. We shall suppose that n is not so large that it exceeds the maximum representable floating-point value, so we can ignore the problem of overflow. We observe that the floating-point mantissa precision must be at least $B+S+N$ bits for the conversion process to work properly.

We define the two components of the twin-float image, $V = (V_1, V_2)$, in an asymmetrical manner respecting the convention that V_1 is the main value while V_2 serves for the heuristic estimate of accuracy. We set V_1 to be the floating point value nearest to n ; in the case of a tie either of the nearest values may be taken. We now set $V_2 = V_1 \pm 2^{e(V_1)-B-S} \rho$ where ρ is an N -bit random value between 1 and 2. Hence the relative difference $(V_2 - V_1)/V_1$ lies between 2^{-B-S-2} and 2^{-B-S} .

Although the values of V_1 and V_2 would be distinguished using a mantissa precision of only $B+S$ bits, we have required a further N bits of precision. These extra bits serve to “guarantee heuristically” that the artificially introduced perturbation exceeds the accumulated rounding error in V_1 by at least a factor of $2^{N/2}$.

Note that the random factor ρ avoids problems of systematic cancellation of the artificial perturbation. For example, if no random factor were used (*e.g.* we replace ρ by 1) then a simple computation like $99/98$ produces an answer whose relative difference between components is smaller by a factor of about 10000 than those artificially added when converting the integers to twin-floats, *i.e.* about 13 bits of the artificial perturbation have been cancelled out.

2.5 Rules of Arithmetic

We can describe the arithmetic most simply in terms of the standard representation — the simple changes to adapt to the alternative representation are left to the reader. The four basic arithmetic operations act independently on the first and second components to produce the respective components of the candidate result, $W = (W_1, W_2)$. If the operation was an addition/subtraction, we conduct a specific check for cancellation (see subsection 2.5.1). In all cases, we check that W is valid as a twin-float, *i.e.* the relative difference between W_1 and W_2 has magnitude at most 2^{-B} ; if the relative difference is too large, the operation produces *failure* — *i.e.* we were unable to obtain a result with at least the minimum requested accuracy. Finally, we check that the relative difference is not too small (see subsection 2.5.2); if it is suspiciously small, we add an artificial perturbation to W_2 . After all checked have passed, we declare W as the actual result.

2.5.1 Cancellation in Addition and Subtraction

In normal exact arithmetic we have an equivalence: given two values α and β , then $\alpha = \beta$ if and only if $\alpha - \beta = 0$. Unfortunately, in twin-float arithmetic we cannot always obtain the same behaviour because there exist twin-float values which are obviously unequal but for which we cannot calculate their difference *with at least the specified minimum accuracy*. Consider this example, with a requested accuracy of 3 decimal digits: α has components (0.999999, 0.999876) and β has components (0.990000, 0.990123); the values are clearly unequal, but we cannot compute $\alpha - \beta$ with 3 decimal digits of accuracy.

So in twin-float arithmetic the relationship between equality and subtraction is a bit more complex — in fact, the only change is that sometimes we can tell that two values are different but we cannot subtract them. Nevertheless we want to retain as much coherence as possible between equality and subtraction. Here is the best we can achieve. The test $\alpha = \beta$ yields *true* if and only if the computation $\alpha - \beta$ produces 0. If $\alpha - \beta$ produces a non-zero value then $\alpha = \beta$ yields *false*. If $\alpha = \beta$ produces *failure* then $\alpha - \beta$ will too. Finally, if $\alpha - \beta$ produces *failure* then $\alpha = \beta$ may produce *failure* or *false*.

The immediate implication is that the test for “perfect cancellation” in addition/subtraction must be identical to the criterion used when the equality test yields *true*.

2.5.2 Not Too Close, Thank You!

The apparently trivial task of dividing a (non-zero) twin-float value by itself proves to be surprisingly delicate. According to the Rules of Arithmetic, we begin by computing a candidate result, $W = (1, 1)$ in this case; note that both components of W are exactly 1. Now, the heuristic accuracy is

determined by the relative difference between the components, which is zero in this case, implying infinite accuracy. So far there are no problems.

If we allow $W = (1, 1)$ as a twin-float value then we can compute $X = W/(W + W + W)$ whose representation as a twin-float is $X = (x, x)$ where x is the underlying floating-point representation of $1/3$. The problem is now evident: since both components of X are identical, we deduce an infinite heuristic accuracy which is no longer a realistic estimate (assuming binary floats). The value X is “dangerous” because it has essentially disabled the heuristic accuracy check, and so subsequent computation with it could well lead to “guaranteed” results which are quite wrong.

We certainly cannot forbid dividing a value by itself (*e.g.* necessary when making a polynomial monic). Instead, we check that the heuristic accuracy of the candidate result is not too high. To be precise we require that the relative difference between the components of W have magnitude at least $2^{-B-S-N/2}$. If the relative difference exceeds this lower limit nothing special happens. Otherwise, if it is too small, then a random perturbation of magnitude roughly 2^{-B-S} is added to the second component; and we go back to repeat the check on the relative difference.

2.6 Conversion of Twin-Floats into Rational Numbers

Acknowledgement: the research reported in this section was conducted in collaboration with Alice Bottaro (at the University of Genoa).

In some cases it is possible to recover an exact rational answer from a computation conducted with twin-float arithmetic. The crucial step is the determination of a suitable rational number given a twin-float. We shall use the inner and outer intervals defined in section 2.2 in the description of the method.

Let us start with some observations. Firstly, like any other operation on twin-floats the conversion may result in *failure*. Next, if the conversion of a twin-float V does succeed (and produces $q \in \mathbb{Q}$) then reconverting q back to a twin-float produces a value equal to V ; *i.e.* the reconversion will succeed and will produce a value W for which the test $V = W$ yields *true*, though it is unlikely that W will be identical to V . Finally, *a priori* there is not a unique choice for q as any two rationals whose relative difference is less than 2^{-B-S-N} are indistinguishable at the precision used for the components of a twin-float.

Since the components of a twin-float V are themselves rational numbers (every float is a rational), we could simply return V_1 as the rational (this surely satisfies the reconversion criterion). So, it is not immediately clear why *failure* should ever need to happen. We opt to allow failure because we want the conversion to give a heuristic guarantee that the rational produced is “clearly the right one”; if there is no obvious unique right choice then the conversion fails.

Returning V_1 as the rational is surely quick and simple, and for some purposes it may be an excellent choice. But in other cases V_1 is likely to be a poor choice because its denominator is restricted to being a power of 2 (or perhaps 10); so, for instance, we could never obtain the rational $1/3$ however high an accuracy we ask for! Yet our main aim is to allow recovery of an exact rational answer without any artificial restriction on denominator (except that implied but the limits of accuracy which we asked for). So we must reject the simple idea of furnishing V_1 as the answer.

We now address the matter of when conversion should succeed, and which of the candidate rational numbers is then chosen. Both answers depend on the definition of the heuristic predicate “clearly the right one”, which we describe right after defining the **bit complexity** of the rational number n/d to be $C(n/d) = \log_2 |nd|$ — it is also a good estimate of the number of bits required to write n/d in binary.

The basic idea is that we want to choose the simplest rational possible. For instance, given a value like 0.33333 we much prefer the answer $1/3$ to the answer $33333/100000$, and in this case $1/3$ is “clearly the right one”. However, with many other values (such as 0.618034) there is no clear choice, so we prefer to let conversion fail rather than give an answer we are unsure of. We quantify this notion of clarity by saying that the (reduced) rational q is “clearly the right one” if it satisfies the reconversion criterion and $C(q) + N/2 < C(q')$ for any other (reduced) rational q' satisfying the reconversion criterion.

We express this in a more algorithmic manner. Let V be the input twin-float. Converting zero is trivial; and if V is negative, we convert $-V$ and then negate the result. So henceforth we assume V is a positive twin-float value. If $\text{inner}(V)$ contains two or more integers, produce *failure* — there is no obviously simplest rational. Otherwise let r be simplest rational in the interval $\text{outer}(V)$. If $r \in \text{inner}(V)$ then return r , other return *failure*.

Determining the simplest rational in an open interval (a, b) can be achieved by converting a and b into continued fractions (see for example [6]). Let $[a_0, a_1, \dots]$ and $[b_0, b_1, \dots]$ be their respective sequences of partial quotients, and let k be the first index where $a_k \neq b_k$. The simplest rational is given by the continued fraction $[a_0, a_1, \dots, a_{k-1}, \alpha]$ where $\alpha = 1 + \min\{a_k, b_k\}$.

The heuristic predicate “clearly the right one” can be fooled if the true rational value we are approximating happens to have a continued fractional partial quotient exceeding $2^{N/2}$ right after the point where we truncated the continued fraction expansions of a and b . Large partial quotients are quite rare, but it is difficult to quantify the likelihood of the predicate making a mistake.

3 Some tricky cases

While twin-float arithmetic does model rational arithmetic well enough for many purposes, it is nonetheless possible to fool the heuristics, and thus to obtain wrong answers. Here are some contrived examples.

3.1 Lack of associativity

It is well-known that floating-point arithmetic does not enjoy all the properties of exact arithmetic, and since twin-floats are built on top of (normal) floats they inherit these defects. For instance, addition is not associative: *e.g.* $(1 + -1) + 10^{-1000} \neq 1 + (-1 + 10^{-1000})$. Similarly, multiplication is not associative: *e.g.* $(2^{1000} \times 2^{1000}) \times 2^{-1000} \neq 2^{1000} \times (2^{1000} \times 2^{-1000})$ in standard “double precision” arithmetic since overflow occurs in one case but not the other. Occasionally even the wide range of `mpf_t` in the GMP library is insufficient: on a 32-bit computer, we encountered problems computing $10^9!$ using the twin-float implementation in CoCoALib.

3.2 Summing values of widely differing magnitude

With limited precision arithmetic a sum of the form $1 + \varepsilon$ poses a problem when ε is small enough: the best approximation to the true answer is 1. This implies that for almost any value x there exists $y > 0$ such that $x + y = x$, an absurdity in exact arithmetic.

In twin-float arithmetic we have the option of producing *failure* instead of a numerical result. So we could declare that summing two values which differ too widely in magnitude will fail. However, this may result in needlessly many failures: obtaining the value 1 as the sum is probably fine unless we subsequently want to compare the result with exactly 1.

In any case for $\varepsilon = -2^{-K}$, we can obtain $1 + \varepsilon$ by summing values of similar magnitude. Start with $s_0 = 2^{-K}$ then successively compute $s_j = s_{j-1} + 2^{j-K}$ for $j = 1, \dots, K - 1$. The final value is $s_{K-1} = 1 + \varepsilon$, but all additions involved numbers differing in magnitude by at most a factor of 2.

3.3 One plus epsilon, again

Consider the sequence a_n defined by $a_0 = \frac{1}{2}$ and $a_n = a_{n-1}(2 - a_{n-1})$ for $n \geq 1$. It is easy to show that $a_n = 1 - 2^{-2^n}$ (in exact arithmetic). So the sequence converges rapidly to 1 but never actually reaches it. Note that we never subtract two values of widely differing magnitude.

If we were to compute the sequence using finite precision floats, at some point we will obtain exactly 1. Twin-floats will behave the same way. It is not clear how some sophisticated finite precision arithmetic could obtain a sensible non-zero value for $1 - a_{100}$, say.

If we specifically wish to add a small number to 1, we proceed as follows. For any $0 < \varepsilon < 1$ we construct a similar sequence whose k -th term is $a_k = 1 - \varepsilon$. Instead of starting from $\frac{1}{2}$ we start from $a_0 = \eta$ where η is the 2^k -th root of ε . Finally, note that $1 + \varepsilon = 2 - a_k$.

4 An Example where failure is a success

This example illustrates one of the perils of computing with limited precision. Consider Muller’s (infamous) recurrence¹ (see p. 48 of [3]):

$$\begin{aligned} a_0 &= 2 \\ a_1 &= -4 \\ a_n &= 111 - 1130/a_{n-1} + 3000/a_{n-1}a_{n-2} \quad \text{for } n > 1 \end{aligned}$$

If we compute this sequence using “double-precision” floating point arithmetic, it converges swiftly to 100. If we try again at higher precision (*e.g.* 100 decimal digits), it converges swiftly again to 100. Indeed, no matter what precision we try, we always get the same result. Intuitively this leads us to believe that we have a guarantee of correctness of the result.

¹I have changed the initial values to integers so that they are exactly representable.

It might come as a surprise then to discover that it is not hard to prove that the sequence a_n converges to 6. But with any finite precision arithmetic explosive growth of a spurious solution deriving from rounding error forces numerical convergence to 100.

One of the example programs supplied as part of CoCoALib shows the behaviour of twin float arithmetic in this computation. Since the arithmetic is of only finite precision, we cannot obtain convergence to 6; the best we can hope for is a failure to converge, and indeed this is what happens: the computation fails due to “insufficient precision”.

5 A Few Benchmarks

Here are a few benchmarks comparing computation times over the rationals with times for the same computation using twin-floats. The column headed “Accuracy” indicates the accuracy requested for the computation. It is not usually obvious what accuracy to request: too low and failure will result, too high and time is wasted handling high precision values. We adopted a simple approach: initially requesting 64 bits, and if failure occurred at some point, we tried again requesting double the accuracy.

The “gin” (generic initial ideal) example shows that twin-float coefficients can lead to enormously faster computing times than with rational coefficients — this is a favourable situation because a generic change of coordinates has to be performed. In contrast the timings show that twin-floats do not always beat rational arithmetic: the computation of the Gröbner basis of “Cyclic7” takes almost four times as long — apparently the rational coefficients manage to stay simple for most of the time, while twin-floats have to use full precision for the entire computation.

Example	Rationals	Twin-Floats	Accuracy
Gin(Cyclic5)	180s	1.9s	128
GB(Cyclic7)	20s	82s	256

All times were obtained using CoCoALib-0.9725 on Macintosh PowerBook G4 (1.7GHz) running MacOSX 10.4.8.

6 Conclusion

Twin-float arithmetic achieves an interesting and useful compromise between the costly but precise rational arithmetic and the quick but unreliable floating-point arithmetic, offering quick computations with heuristically verified accuracy. The usefulness in Gröbner basis computations had already pointed out by Traverso and Zanoni. The implementation in CoCoALib is as a `ring`, and so can be applied to a wide range of computations, including Gröbner bases.

References

- [1] F Roullier, P Zimmermann “Efficient Isolation of Polynomial Real Roots” *INRIA Rapport de Recherche 4113*, Feb 2001.
- [2] C Traverso, A Zanoni “Numerical stability and stabilization of Gröbner basis computation” *Proc. ISSAC 2002 (Lille)*, pp. 250–257, July 2002.
- [3] J M Muller, *Arithmetique des Ordinateurs*, Masson, Paris, 1989; the book is also available online at <http://prunel.ccsd.cnrs.fr/ensl-00086707>
- [4] The CoCoA web site <http://cocoa.dima.unige.it/>
- [5] The GMP web site <http://www.swox.com/gmp/>
- [6] G H Hardy, E M Wright, *An Introduction to the Theory of Numbers*, (fifth edition), Oxford University Press, 1979