Randomized Interval Analysis Checks for the Equivalence of Mathematical Expressions

Travis W. Fisher
John L. Orr
Stephen D. Scott

The problem of determining whether two mathematical expressions are equivalent is very difficult, and even algorithmically undecidable in complete generality. Presented here are two algorithms that use interval analysis to make quick and effective probabilistic comparisons of expressions. These algorithms benefit from a novel extension of the standard definitions of interval analysis, to better deal with functions that do not exist everywhere. The algorithms are based on random sampling and are applicable to a wide range of functions, can be carried out at high speed, and admit only one-sided error in the sense that equivalent expressions are never judged unequal. The first algorithm tests for equivalence of expressions and the second algorithm tests for equivalence of expressions up to an additive constant. These algorithms have been found highly accurate in computer grading of mathematics exams, and have potentially wider applicability.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Algorithm design and analysis

Additional Key Words and Phrases: zero equivalence, functional equivalence, interval analysis

1. INTRODUCTION

The problem of determining unequivocally whether one mathematical expression is equivalent to another expression is very difficult, and even algorithmically undecidable in complete generality. This problem is known to be undecidable for the class of expressions built up from the constants 1 and π, the variable x, and the functions abs (absolute value) and $\sin$ using arithmetic operations and composition of functions [Richardson 1968]. Note that expressions f and g are equivalent if and only if the expression $f - g$ is equivalent to zero; for this reason the question is often referred to as the *zero equivalence* problem.

In this paper we describe practical algorithms to deal with the zero equivalence problem and the closely related problem of determining if the difference between two expressions is a constant. We have identified three properties that are desirable in an algorithm to compare two given expressions:

(1) The algorithm should be able to compare arbitrary expressions involving a large class

Stephen Scott was supported in part by NSF Grant CCR-9877080 with matching funds from CCIS and a grant from the Layman Foundation.
Name: Travis Fisher
Address: Dept. of Mathematics & Statistics, University of Nebraska, Lincoln, NE 68588-0323.
Current address: Dept. of Mathematics, The Pennsylvania State University, University Park, PA 16802
Name: John Orr
Address: Dept. of Mathematics & Statistics, University of Nebraska, Lincoln, NE 68588-0323.
Name: Stephen Scott
Address: Dept. of Computer Science & Engineering, University of Nebraska, Lincoln, NE 68588-0115.

of operations[1].

(2) Comparisons need to take place quickly: for real-time applications comparisons should not take more than a fraction of a second.

(3) The algorithm should be guaranteed to produce correct results for each comparison.

Unfortunately, in view of the above mentioned undecidability results, it is not possible to attain all three of these properties. The algorithms that we present settle for a weaker version of the third property: the algorithm will only be guaranteed to have returned a correct result when its return value indicates that the expressions under consideration are *unequal*. This property is often referred to as "one-sided error."

This limitation to one-sided error is satisfactory in many applications. Indeed, if the goal of the application is to disprove purported identities, then the value of these algorithms lies in the complete confidence one gains for those identities that the algorithms disprove. From a practical standpoint, our algorithms have also proven effective for verifying identities in an implementation involving a broad class of functions. In an implementation where a guarantee of correct results is critical, the algorithms we present could be used in tandem with methods that guarantee only one-sided error in the opposite direction. We have used this technique to evaluate the performance of our implementation of these algorithms (see Section 4.1).

1.1 Statement of Problem

By an *expression* we mean a well-formed string of symbols which represents a real-valued function dependent on zero or more variables. An expression that contains n variables represents an n-place function in the obvious way. At those points where an expression is not defined in the real numbers, we will say that the expression takes the value *NaN* ("Not a Number"). Throughout this document we will use the phrases "the expression is undefined" and "the expression takes the value *NaN*" interchangeably. We will use $\mathbf{R}$ to denote the set of real numbers and $\mathbf{R}^\dagger$ to denote the set $\mathbf{R} \cup \{NaN\}$. Two expressions will be considered equivalent if and only if they take the same value in $\mathbf{R}^\dagger$ for each possible real assignment to the variables.

The main question we are concerned with is how to determine if two expressions represent the same function. For example, `3 sin(y) + cos(x)` and `cos(2 pi - x) - 3 sin(-y)` are each valid expressions which do, in fact, represent the same function. For any assignment to the variables `x` and `y`, each expression will evaluate to the same result. The expressions `ln(x)` and `ln(abs(x))`, on the other hand, do not represent the same function. For any negative `x` the latter expression evaluates to a real number and the former evaluates to *NaN*.

2. ALGORITHMS

2.1 Overview of Interval Analysis

Interval mathematics is an extension of the mathematics of the real line. We will start with the set of closed intervals of the real line and introduce methods of arithmetic with these intervals. A computation involving two intervals produces a third interval whose bounds

[1] The implementation that led to this paper evaluates expressions built in an arbitrary way from the operations `+`, `-`, `*`, `/`, `^`, and the functions `sin`, `cos`, `ln`, `abs`, and `arcsin`.

are bounds on the possible values of the result of that computation. In exact interval mathematics, the result of the computation would be the minimal bounding interval. Practical implementations carry out *rounded* interval mathematics, where the resulting interval is slightly larger than a minimal bounding interval but is still guaranteed to contain all possible values of the result of the computation. The basics of the subject were first laid out by Moore [1966] and can be found in many more recent expositions; here we give only a minimal description of the subject and introduce the notation used in the remainder of this paper.

DEFINITION 1. Let $[x_1, x_2]$ denote the set $\{x \in \mathbf{R} \mid x_1 \leq x \leq x_2\}$. The set of *real intervals* is the set $\mathcal{I}(\mathbf{R}) = \{\, [x_1, x_2] \mid x_1, x_2 \in \mathbf{R}\}$.

DEFINITION 2. For $A, B \in \mathcal{I}(\mathbf{R})$, and for $\circ$ one of the operations $+,-,\times,/$, or $\hat{}$

$$A \circ B = \{\, a \circ b \mid a \in A \text{ and } b \in B\}$$

provided that $a \circ b$ exists for all a in A and all b in B.

DEFINITION 3. For continuous functions $f : \mathbf{R}^n \to \mathbf{R}$, the *natural interval extension* of f is defined by $f(A_1, A_2, \ldots, A_n) = \{\, f(a_1, a_2, \ldots, a_n) \mid a_i \in A_i\}$ for $A_i \in \mathcal{I}(\mathbf{R})$, $i = 1, 2, \ldots, n$.

Note that an interval, as well as the cross-product of n intervals, is a connected set. The continuous image of a connected set is connected, so the result of applying the interval extension of a function is again an interval. Furthermore, since the operations $+,-,\times,/$, and $\hat{}$ are continuous, Definition 2 is subsumed by Definition 3.

The property of interval arithmetic that is most essential to our purposes is known variously as *inclusion monotonicity* [Moore 1966] or *inclusion isotonicity* [Hammer et al. 1995], and guarantees that the result of a computation carried out on an interval I will be a superset of the result of the computation carried out on a subinterval of I.

DEFINITION 4. Let $A, B \in \mathcal{I}(\mathbf{R})$. The binary operation $\circ$ defined on $\mathcal{I}(\mathbf{R})$ is *inclusion monotonic* if $A \circ B \subset A' \circ B'$ whenever $A', B' \in \mathcal{I}(\mathbf{R})$ satisfy $A \subset A'$ and $B \subset B'$. The function $f : \mathcal{I}(\mathbf{R}) \to \mathcal{I}(\mathbf{R})$ is inclusion monotonic if $f(A) \subset f(A')$ whenever $A' \in \mathcal{I}(\mathbf{R})$ satisfies $A \subset A'$.

It is easy to see from Definition 3 that the natural interval extension of a continuous function is inclusion monotonic. For if $f : \mathbf{R}^n \to \mathbf{R}$ and $A_i \subset A_i'$ are real intervals then $f(A_1, \ldots, A_n) = \{f(a_1, \ldots, a_n) \mid a_i \in A_i\} \subset \{f(a_1, \ldots, a_n) \mid a_i \in A_i'\} = f(A_1', \ldots, A_n')$. Since Definition 2 is a special case of Definition 3, the operations $+$, $-$, $\times$, $/$, and $\hat{}$ are also inclusion monotonic on $\mathcal{I}(\mathbf{R})$.

We use intervals to represent upper and lower bounds on the possible values of a real-valued expression. By making copious use of the inclusion monotonicity property of interval operations and functions, it is straightforward to calculate such upper and lower bounds even for very complicated expressions. We can keep our bounds absolutely accurate even if our methods of calculation introduce error (due to rounding in finite-precision arithmetic) at each step. As long as we have a bound on the error introduced at any one step, we can make use of inclusion monotonicity to expand our intervals to take that error into account. If done carefully, the interval that we produce as the result of a calculation is guaranteed to contain the correct value of that calculation.

DEFINITION 5. Let the set of *machine numbers* M be the set of numbers representable in a particular implementation (e.g. the set of double-precision floating point numbers). The set of *machine intervals* is the set $\mathcal{I}(M) = \{\, [x_1, x_2] \mid x_1, x_2 \in M \}$.

An *acceptable* rounded interval arithmetic scheme will be one in which the result of every operation is a machine interval that contains the result of the operation in the real intervals. Thus an acceptable rounded interval arithmetic scheme must maintain the property of inclusion monotonicity. An *optimal* rounded arithmetic scheme would be one in which the result of every operation was the smallest machine interval containing the result of the operation in the real intervals.

Note that in order to implement an acceptable scheme of rounded interval arithmetic, each real number must be contained in some member of $\mathcal{I}(M)$. Using floating point arithmetic as described in the IEEE 754 [1985] standard, this is achieved by means of special machine number representations for positive and negative infinity. These special values are considered to be part of the set M and are used to produce intervals that contain values larger than any finite floating point number.

It is straighforward to implement an acceptable rounded interval arithmetic scheme on any system where the desired operations are already implemented with a known degree of accuracy. We denote such rounded machine operations by the use of a subscripted M. Let $+_M$, $-_M$, $\times_M$, $/_M$, and $\hat{}_M$ be binary operations defined on $M \times M$ by the computer arithmetic operations that produce mathematically correct results with the possible exception of rounding taking place in the unit in the last place (ULP). These are operations from $M \times M$ to $M \cup \{NaN\}$, where *NaN* occurs as a result of division by zero, exponentiation of a negative number to a fractional power, or exponentiation of zero to the zeroth power. For a quantity in M, let underlines, $\underline{x}$, and overlines, $\overline{x}$, represent respectively the operations of subtracting and adding one ULP to the quantity x. It is easy to see, for instance, that if $A = [a_1, a_2]$ and $B = [b_1, b_2]$ are both intervals in $\mathcal{I}(M)$ then

$$A \times_M B = [\min\{\underline{a_1 \times_M b_1}, \underline{a_1 \times_M b_2}, \underline{a_2 \times_M b_1}, \underline{a_2 \times_M b_2}\}, \\ \max\{\overline{a_1 \times_M b_1}, \overline{a_1 \times_M b_2}, \overline{a_2 \times_M b_1}, \overline{a_2 \times_M b_2}\}]$$

defines an acceptable, but not optimal, implementation of rounded interval multiplication.

The general principle involved in defining an easily computable form of a rounded interval version of an elementary function is that one need only examine a finite number of points to determine good upper and lower bounds on the function's value. It is important to recall that we have no desire to hand-create rounded interval versions of every function $f : \mathbf{R}^n \to \mathbf{R}$; we just need to create rounded interval versions of the elementary functions. For example suppose that $\sin_M : M \to M$ is a machine implementation of the sin function that is free from computational error. Suppose that $A = [a_1, a_2]$ is an interval in $\mathcal{I}(M)$. We will define the interval version of the sin function in a number of cases, always being careful to maintain the property of inclusion monotonicity. If sin has (or might have) any local maxima on A, then the upper limit of $\sin_M(A)$ is 1. Otherwise the upper limit is the maximum of $\overline{\sin_M(a_1)}$ and $\overline{\sin_M(a_2)}$. If sin has (or might have) any local minima on A, then the lower limit of $\sin_M(A)$ is -1. Otherwise the lower limit is the minimum of $\underline{\sin_M(a_1)}$ and $\underline{\sin_M(a_2)}$.

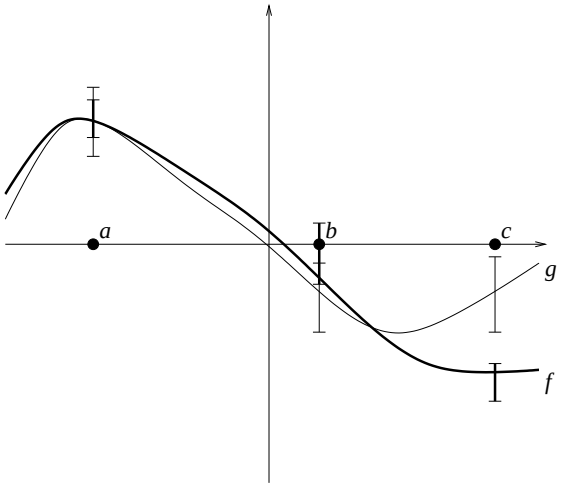

Fig. 1. Interval comparison of functions. The curves represent the actual values of $f(x)$ and $g(x)$; the error bars show interval evaluations of f and g at the values a, b, and c. Note that the intervals overlap at points a and b, so the functions are compatible at those points. At point c, however, the intervals do not overlap so that sample shows the functions are not equal.

2.2 Interval Solution of the Zero Equivalence Problem

The procedure we use to test if two expressions f and g are equivalent using interval arithmetic is carried out as a series of trials. Each trial begins by assigning random values to the variables in the expressions f and g. The expressions are then evaluated using rounded interval arithmetic. In the ordinary case, the evaluation produces a machine interval for each expression. If these machine intervals do not overlap, the expressions are judged to be unequal and trials stop. Otherwise, more random trials are carried out. After some preset number of trials have been carried out in which all the intervals obtained overlap, the expressions are judged to be equal. An illustration of the interval comparison is shown in Figure 1.

A key thing to observe about the above process is that with interval arithmetic, unlike ordinary floating point arithmetic, there is no danger of mistakenly declaring a pair of expressions to be unequal. The property of inclusion monotonicity guarantees that, when f is evaluated at a point, the resulting interval *must* contain the correct value of f at that point. The same holds for the interval produced for the expression g at that same point. Then if f and g are equivalent expressions, the intervals produced by each must overlap at that point. If any point is found where f and g produce intervals which do not overlap, the expressions have been conclusively shown to be unequal. This property ensures that an interval evaluation scheme has only one-sided error. While unequal expressions may sometimes be close enough in value to be judged correct, equivalent expressions will never be mistakenly judged to be unequal.

There is one important point not yet addressed. We need to be careful of what happens when one of the expressions is undefined. For example, take the division operator. When we defined the real interval operation of division, we left $A/_M B$ undefined in the case where $0 \in B$. In fact there are three cases to consider for $A/_M B$: The first case is the one that we have been considering all along: that $0 \notin B$. In this case, the operation $A/_M B$ produces a

Algorithm 1.
start with TRIALS equal to 0
repeat until TRIALS $>$ MAXTRIALS
  assign random values to each variable in f and g
  let U be the rounded interval evaluation of f under those assignments
  let V be the rounded interval evaluation of g under those assignments
  if $U \in \mathcal{I}(M)$ and $V \in \mathcal{I}(M)$ and $U \cap V = \emptyset$
    return FALSE (we have found a miss)
  else if $U =$ *Certainly NaN* and $V \in \mathcal{I}(M)$
    return FALSE (we have found a miss)
  else if $U \in \mathcal{I}(M)$ and $V =$ *Certainly NaN*
    return FALSE (we have found a miss)
  increment TRIALS
return TRUE (if we cannot demonstrate that f and g differ, assume they are equal)

Fig. 2. Probabilistic rounded interval check if expressions f and g are equivalent.

machine interval. The second case is that $B = [0, 0]$. In this case, the quotient $a/b =$ *NaN* for all $a \in A$ and $b \in B$. We will describe this as $A /_M B =$ *Certainly NaN*. The third case is that $0 \in B$, but B also contains nonzero values. In this case, there are some combinations of $a \in A$ and $b \in B$ such that $a/b \in \mathbf{R}$. We will describe this as $A /_M B =$ *Possibly NaN*. Taken together these three cases allow the division operation $A /_M B$ to be defined for all $A, B \in \mathcal{I}(M)$ as a map from $\mathcal{I}(M) \times \mathcal{I}(M)$ to $\mathcal{I}(M) \cup \{$*Certainly NaN*$\} \cup \{$*Possibly NaN*$\}$.

The same three cases can be applied to the natural machine interval extension of any function whose value does not always exist in the real numbers. For example, consider the square root function as a function from $\mathbf{R}$ to $\mathbf{R}^\dagger$. For any real $x \geq 0$ we have $\sqrt{x} \in \mathbf{R}$. For $x < 0$ we have $\sqrt{x} = NaN$. Then for an interval $A \in \mathcal{I}(M)$, if A contains only nonnegative values we have case one: $\sqrt[M]{A} \in \mathcal{I}(M)$. If A contains only negative values we have case two: $\sqrt[M]{A} =$ *Certainly NaN*. If A contains both negative and nonnegative values we have case three: $\sqrt[M]{A} =$ *Possibly NaN*.

When we take into account the possibilities of *Certainly NaN* and *Possibly NaN*, the procedure to test if two expressions f and g are equivalent remains essentially unchanged. One has only to consider more possibilities to decide what the outcome of a trial is. Random values are assigned to each of the variables in the expressions f and g, and the expressions are evaluated using interval mathematics. We will call a trial a "miss" if that trial shows that f and g cannot be equivalent. If both f and g produce an interval, the trial is a miss if the intervals do not overlap. If one of the expressions produces the result *Certainly NaN* and the other produces an interval, then the trial is a miss. In each of the other possible cases, the trial is inconclusive. This algorithm is shown in Figure 2.

2.3 Interval Solution of Expression Equivalence Modulo a Constant

Given two expressions f and g, we sometimes want to determine if there exists a constant C such that $f = g + C$. This question arises naturally when trying to check identities involving the indefinite integral of a given expression. Different antiderivatives of an expression differ by an additive constant.

Interval mathematics provides an elegant solution to the problem of probabilistically determining whether two expressions f and g are equivalent modulo a constant. One can perform a series of random trials, just as in the case of determining equivalence of expres-

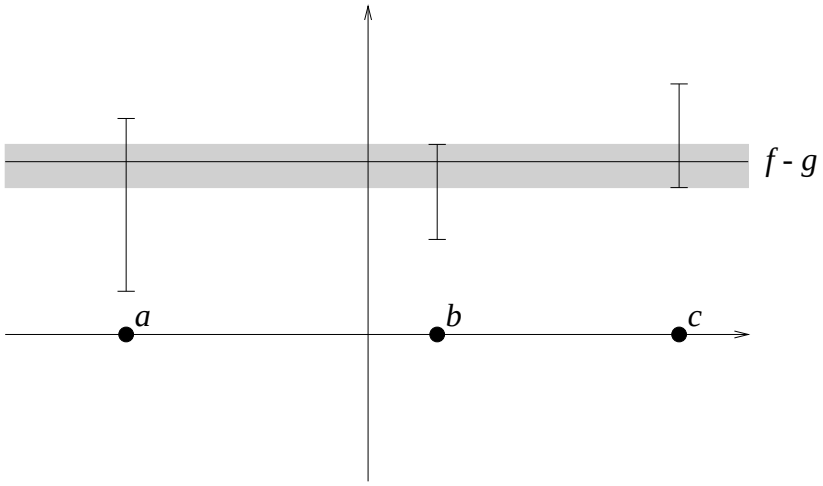

Fig. 3. Interval comparison of functions modulo a constant. Intervals are evaluated for $f - g$ at sample points a, b, and c. The shaded area represents the intersection of these intervals. As long as that intersection is non-empty, it is possible that f and g differ by a constant.

sions. For each trial, a random value is assigned to each variable in the expressions f and g. The expressions are then evaluated. If both expressions produce a machine interval as the result of the trial, then these intervals are subtracted and the interval representing their difference is recorded. If the intersection of this difference interval with all previous difference intervals is empty, the trial is a miss. If one of the expressions produces the result *Certainly NaN* and the other produces an interval, then the trial is a miss. In each of the other possible cases, the trial is inconclusive. As soon as a miss is found, the expressions are judged to be not equivalent. After enough trials, if no miss has been found the expressions are judged to be equivalent. The interval comparison is illustrated in Figure 3 and the algorithm is detailed in Figure 4.

This process also has the nice property of having only one-sided error. Two expressions that are equivalent modulo a constant are never judged to be inequivalent. To see this, suppose that f and g are expressions such that $f = g + C$ for some constant $C \in \mathbf{R}$. By the property of inclusion monotonicity of interval mathematics, the interval result of $g - f$ must contain C wherever f and g exist. Thus the intersection of all the differences $g - f$ must still contain C and therefore be nonempty. Furthermore, whenever f exists, g also exists and whenever g exists, f also exists. Thus if one evaluates to *Certainly NaN* the other cannot evaluate to an interval. This takes care of both possible ways in which a trial could produce a miss, so two expressions equivalent modulo a constant will not produce any misses and not be judged unequal.

2.4 Limitations

As we noted in Section 1, any algorithm that attempts to solve the expression equivalence problem must make a sacrifice either in the class of expressions that can be compared or in the accuracy with which this comparison takes place. Our algorithms have the possibility of one-sided error, in that there are inequivalent expressions which our algorithms will wrongly judged to be equal.

Algorithm 2.
start with TRIALS equal to 0 and INTERSECTION equal to $[-\infty, \infty]$
repeat until TRIALS $>$ MAXTRIALS
  assign random values to each variable in f and g
  let U be the rounded interval evaluation of f under those assignments
  let V be the rounded interval evaluation of g under those assignments
  if $U \in \mathcal{I}(M)$ and $V \in \mathcal{I}(M)$
    let INTERSECTION equal INTERSECTION $\cap$ $(U -_M V)$
    if INTERSECTION $= \emptyset$
      return FALSE (we have found a miss)
  else if $U =$ *Certainly NaN* and $V \in \mathcal{I}(M)$
    return FALSE (we have found a miss)
  else if $U \in \mathcal{I}(M)$ and $V =$ *Certainly NaN*
    return FALSE (we have found a miss)
  increment TRIALS
return TRUE (there is still a range of constants by which f and g might differ)

Fig. 4. A rounded-interval algorithm for comparison of expressions f and g up to an additive constant.

One source of this kind of error comes from using the set M of machine numbers as the domain and range of the evaluation of expressions. Any finite set M of machine numbers must have, disregarding representations for positive and negative infinity, a largest and a smallest element. A pair of expressions that differ only outside of the domain M cannot be recognized as unequal by our algorithms. Similarly, a pair of expressions that agree to within one ULP (or in practice, a few ULPs) cannot be distinguished by our algorithms. All values larger than the largest finite member of M are equivalent, as are all values smaller than the smallest nonzero member of M. Consequently expressions that take only sufficiently large or sufficiently small values cannot be distinguished.

Another source of error is that our algorithm will have trouble distinguishing a numerically ill-conditioned expression from other expressions. In normal floating point arithmetic, the evaluation of an ill-conditioned expression is numerically unstable because of the accumulated effect of rounding errors in floating point calculations. In interval arithmetic, these errors are contained inside bounding intervals, so an ill-conditioned expression produces a very large interval as its result.

Finally, we should note that a practical implementation of our algorithms must make a trade-off between speed and accuracy. While the algorithms can theoretically distinguish between any pair of expressions which differ by a minimal number of ULPs for some values in the domain M, in order to have a usable implementation we need to limit ourselves to sampling a relatively small number of points from M. If a pair of expressions are only detectibly different on a portion of the domain M, the liklihood of detecting that difference depends on how many trials are carried out and how the sample point for each trial is chosen[2].

[2] In our implementation, we found that sample points close to zero were more likely to uncover a detectible difference, and we decided to chose sample points in a Gaussian distribution around zero. The most effective distribution of sample points obviously depends on the expressions which are under comparison.

3. OTHER APPROACHES

We give here a brief description of some of the other practical methods to solve the zero equivalence problem described in the literature and compare these to our approach. There is also a body of literature on the theoretical basis of this problem; for more details on this side of the subject see Shackell [1993] and the references contained therein.

3.1 Pure Floating Point Approach

The interval-based algorithms of Section 2 are a direct descendent of a simpler floating point approach in which the values of two expressions are directly compared at a random sampling of points. Without use of interval analysis, however, it is difficult to decide when the results of two floating point computations should be considered equal. Small floating point discrepancies are utterly routine. For instance even $0.1 + 0.2$ is not equal to 0.3 in double-precision floating point arithmetic (as described in the IEEE 754 [1985] standard). In expressions that are ill-conditioned, two slightly different presentations of the same underlying expression could produce arbitrarily different floating point results.

To get reasonable results from this simple approach, one can judge floating point results equal if they match to within a small, arbitrary tolerance. Unfortunately, this method has no error guarantees whatsoever. Unequal expressions may be judged to be equivalent and equivalent expressions may be judged not to be so, with the relative probabilities of these errors depending on the choice of the tolerance allowed.

3.2 Finite Field Probabilistic Approach

A somewhat similar probabilistic method is described in Gonnet [1984, 1986], which builds on the work of Martin [1971]. The fundamental idea is similar to ours: a series of trials is carried out, where each trial involves evaluating both expressions at a sample point. Each trial either conclusively demonstrates that the expressions differ or fails to do so. This method produces one-sided error in the same direction as our approach.

The key difference of this approach from ours is that evaluation of the expressions, rather than being carried out in floating point arithmetic, is carried out in modular arithmetic. It is easy to see how this works for expressions containing only addition, subtraction, multiplication, and division: arithmetic carried out modulo a prime p is still arithmetic in a field. The method can be extended fairly well to work with exponentiation, and can be stretched to simulate the behavior of some transcendental functions in modular arithmetic.

However, the class of functions that this approach can compare is quite restricted. For instance, suppose a and b are expressions. It can be shown through an application of Fermat's Little Theorem that in order to compute a^b modulo a prime p, b needs to be computed in modular arithmetic modulo $p - 1$. The integers modulo $p - 1$ are not a field, since $p - 1$ is even. In particular, $b = 1/2$ is always undefined since there is no inverse of 2 modulo an even number. This causes great difficulty in dealing with square roots. Another basic problem is that transcendental functions can only be simulated and not properly represented. These simulations are troublesome, requiring consideration of various esoteric special cases and suffering from the difficulty that results are not always defined. It seems unavoidable that many of the expressions that occur naturally cannot be compared by this method.

3.3 Symbolic Simplification

A completely different approach to the zero equivalence problem is to manipulate the expressions symbolically. For some classes of expressions one can define a standard form and manipulate each expression into that form. For example, expressions involving only addition, subtraction, and multiplication have a familiar standard form – these are simply polynomials. To compare if two such expressions are equivalent, one can multiply them out and compare the coefficients of matching terms. For more complicated expressions (e.g. those involving trigonometric functions) the task of defining a standard form becomes more difficult or impossible.

One caveat is that the difficulty of arithmetic remains even in symbolic manipulation. For instance the multiplication and addition involved to put a polynomial into standard form would suffer from debilitating inexactness if done using floating point arithmetic. One way around this is to avoid floating point or rounded arithmetic completely, instead using arbitrary-precision decimals or rationals and dealing with irrational numbers only symbolically.

For classes of expressions for which there is no standard form, the symbolic simplification approach tends to produce a one-sided error evaluation in the opposite direction than the randomized interval approach produces. If through manipulations and substitutions one expression can be made to match the other expression, the expressions are conclusively shown to be equivalent. If, however, the expressions cannot be made to match through a series of manipulations, in general it is not necessarily safe to conclude that the expressions are not equivalent.

4. IMPLEMENTATION

The algorithms of Section 2 were implemented for use in computer-based mathematics exams. These exams are delivered over the WWW by a system called *eGrade*, developed by the second author. Earlier versions of this software [Orr 1998; Orr 1999] used a version of the floating point algorithm described in Section 3.1 to compare student responses on a question to the correct answer. The results of this paper come from an effort to improve on that algorithm.

We have implemented a variation of Algorithms 1 and 2 in Java for use in *eGrade*. The interval mathematics methods use a modified version of the *ia_math* package [Hickey 1997]. Due largely to the numerical untrustworthiness of various implementations of the Java Virtual Machine, our implementation cannot completely guarantee the theoretical one-sided error properties we describe[3]. However, despite the fact that some Java implementations have been seen to commit some mathematical atrocities, we have yet to see the one-sided error property violated in practice.

4.1 Evaluation

We have made an effort to evaluate the effectiveness of Algorithm 1 and 2 in the context its implementation in *eGrade*. This system has been in use in the University of Nebraska-Lincoln for a number of semesters, and during this time details have been recorded of

[3]Those properties should be achievable, however, in an environment where the numerical accuracy of computations is more carefully controlled. Reliable interval mathematics libraries exist for several languages and platforms, taking advantage of modern hardware's compliance with the IEEE 754 standard.

exams taken. This gives us a database of tens of thousands of actual student responses that can be used in order to determine how effective our algorithm is in practice.

We evaluated Algorithm 1 on a set of over 8,000 student responses to calculus questions involving computing derivatives. The student responses were compared to the correct expressions using both our algorithm and a symbolic simplification method[4] using Maple [Waterloo Maple Inc. 1996]. Since these approaches have one-sided error in opposite directions (see Section 3.3), whenever the algorithms agree they must both be correct. We found only 3 responses out of our sample of 8,000 where the methods disagreed. In each case our approach was hand-verified to be correct.

We evaluated Algorithm 2 on a set of over 4,000 student responses to calculus questions involving computing indefinite integrals. The responses were again checked using both our algorithm and Maple's symbolic simplification method. The success of this evaluation depended upon the fact that in the student responses, separation constants other than 0 are rare enough that we could examine those cases by hand. Hand-verification of discrepancies revealed that our algorithm accepted four incorrect responses as correct. The difference between the incorrect responses and correct answers was that in two cases 0.3333333334 had been substituted for $1/3$ and in two cases 0.6666666667 had been substituted for $2/3$ in complicated expressions. Mathematically, those expressions were not equivalent modulo a constant, though with the precision of our computations they were numerically indistinguishable.

4.2 Speed

It is clear from a quick examination of our algorithms that the worst-case time to completion is directly proportional to the number of trials that must be carried out before termination conditions are reached. In order to speed up our implementations of Algorithms 1 and 2, we modified the termination conditions while still preserving the one-sided error property. Originally, in Algorithm 1, trials continued until definite evidence of inequivalence was found or a fixed number (MAXTRIALS) of trials was exceeded. Our modified implementation counts a "hit" for each trial in which the expressions evaluate to intervals that overlap. When a certain number of hits (MAXHITS) are achieved, this is considered sufficient evidence to judge the functions equivalent.

By setting MAXHITS to a relatively small value while leaving MAXTRIALS at a large value, we achieve a balance between speed and accuracy. The common cases, where either the expressions are found to disagree within a few trials or else accumulate a sufficient number of hits within a few trials, are handled quickly. In the rarer cases where almost every evaluation produces Possibly NaN, the large value of MAXTRIALS allows testing to continue, increasing the likelihood that some trial will prove the expressions inequivalent or that sufficient hits will accumulate to provide a strong indication that the expressions are equivalent.

We decided on the basis of our empirical tests that setting MAXHITS to 14 gave a reasonable balance between speed and effectiveness of catching incorrect expressions for our particular application. Using this low number of trials did not affect the results of our tests of Algorithm 1, and introduced only one additional error into the results of our tests of Algorithm 2. Our implementation had an average response time of 0.031 seconds spent

[4]Namely, we compared expressions f and g by checking if `evalb(simplify(f-g)=0);` returned *true* or *false*.

to evaluate each pair of expressions, running under a standard Java interpreter on a 200 MHz Pentium machine.

5. CONCLUSIONS

We have developed two closely related algorithms. The first of these provides a randomized solution to the problem of determining whether or not two expressions are equivalent. The second of these provides a randomized solution to the problem of determining whether or not two expressions differ by only an additive constant. Each of these algorithms allows only one-sided error (provided it is implemented on a machine that does not suffer from computational error in its elementary math routines). We have shown how the one-sided error property arises from the inclusion monotonicity property of interval mathematics. In the process, we have developed a careful treatment of rounded interval mathematics employing the conditions we call *Certainly NaN* and *Possibly NaN*. The use of these specific conditions is original, although probably closely related to other schemes that have been introduced to extend interval mathematics.

REFERENCES

GONNET, G. H. 1984. Determining equivalence of expressions in random polynomial time. In *Proceedings of the 16th ACM Symposium on the Theory of Computing* (April 1984), pp. 334–341.

GONNET, G. H. 1986. New results for random determination of equivalence of expressions. In *Proceedings of the 1986 ACM Symposium on Symbolic and Algebraic Computing* (July 1986), pp. 127–131.

HAMMER, R., HOCKS, M., KULISCH, U., AND RATZ, D. 1995. *C++ Toolbox for Verified Computing.* Springer-Verlag, Berlin.

HICKEY, T. J. 1997. ia_math library version 0.1beta1. http://www.cs.brandeis.edu/~tim/Packages/ia/ia_math/.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. 1985. *IEEE Standard for Binary Floating-Point Arithmetic.* IEEE.

MARTIN, W. A. 1971. Determining the equivalence of algebraic expressions by hash coding. *Journal of the ACM 18,* 4 (October), 549–558.

MOORE, R. E. 1966. *Interval Analysis.* Prentice-Hall.

ORR, J. L. 1998. *Wiley Web Tests for Calculus.* John Wiley & Sons, New York.

ORR, J. L. 1999. *Wiley Web Tests for Precalculus & College Algebra*. John Wiley & Sons, New York.

RICHARDSON, D. 1968. Some undecidable problems involving elementary functions of a real variable. *Journal of Symbolic Logic 33*, 4 (December), 514–520.

SHACKELL, J. 1993. Zero-equivalence in funciton fields defined by algebraic differential equations. *Transactions of the American Mathematical Society 336*, 1 (March), 151–171.

Waterloo Maple Inc. 1996. *Maple V Release 4.* Waterloo Maple Inc.