

# Probabilistic Checks for the Equivalence of Mathematical Expressions

A Senior Thesis

by

Travis Fisher

Presented to the College of Arts and Sciences

and

the University of Nebraska-Lincoln Honors Program

In Partial Fulfillment of the Requirements

For a Degree with Distinction

and

Graduation from the Honors Program

Under the supervision of Dr. John Orr and Dr. Stephen Scott

The University of Nebraska-Lincoln

March, 1999

## Abstract

This thesis presents work that has been done to develop, implement, and evaluate routines to probabilistically check the equivalence of mathematical expressions within an online . The algorithms that we have developed make use of a rounded interval mathematics scheme that is not new, but that has not previously been applied to problems of this kind. This work has resulted in significant improvement over the algorithms that were previously being used.

In this thesis, we will present the background material necessary to motivate and understand the interval mathematical algorithms we develop, and to understand the behavior of the Java implementation of these algorithms. The primary original contribution of this thesis work is the development and implementation of two closely related algorithms. The first of these provides a probabilistic solution to the problem of determining whether or not two mathematical expressions are equivalent. The second of these provides a probabilistic solution to the problem of determining whether or not two mathematical expressions differ by only an additive constant. Each of these algorithms uses methods of rounded interval mathematics to control for errors of floating point arithmetic. We will show theoretically that these algorithms can be made to allow only one-sided error when implemented correctly. Another contribution of this thesis is the development of a treatment of rounded interval mathematics which makes a careful distinction between different ways in which an interval function can fail to exist. The use of the specific conditions that we describe is original, although probably closely related to other schemes that have been introduced to extend interval mathematics.

The algorithms that we describe here have been implemented and are currently in use within the Gateway Web Testing System in the University of Nebraska-Lincoln Math Department.

## Preface

This thesis is primarily a result of work carried out in the Fall semester of 1998 and the Spring semester of 1999, during which time I have been an undergraduate student at the University of Nebraska-Lincoln (UNL). My work with Dr. Orr on the Gateway Web Testing System extends farther back than that; I wrote some routines for the system in the Spring of 1998 and worked intensely for a few weeks on part of the system in the Summer of 1998.

The primary area of this thesis is Computer Science and the secondary area of this thesis is Mathematics. I have tried to make this thesis accessible and interesting to readers whose primary background is in either one of those areas.

I would like to extend my utmost gratitude to both of my advisors on this project. Both Dr. Orr and Dr. Scott have been exceedingly helpful and patient throughout the process. Dr. Orr especially has worked with me through all of the details, and without his encouragement and advice the project would have gone nowhere. (This is not to mention the fact that without his Gateway System, the project would not exist in the first place.)

# 1 Introduction

## 1.1 Overview of the Project

Over the last two years, John Orr has been developing a package to carry out automated student assessment via the World Wide Web. This system has been used extensively by the UNL Math Department to carry out skill mastery tests (“Gateway Exams”) for precalculus and calculus classes, and by a variety of other departments. The package is being published by John Wiley & Sons as a companion to their calculus book series [11] [12]. It is written entirely in Java, including its own web server, CGI engine, exam development, administration, and evaluation tools. Large question databases have been developed for precalculus and calculus, and smaller databases have been developed in other areas.

One of the features that sets the gateway exam system apart from other online testing systems is the fact that it allows students to enter free-form responses to questions. For example, a student taking a gateway calculus exam may be asked to enter the indefinite integral of a given function. She will be given a text entry box in which to type an arbitrary formula in terms of simple functions, constants, and arithmetic operators. When her answer is submitted, the system needs to determine whether or not the response is correct. Such a determination needs to be carried out as quickly and as accurately as possible.

The problem of determining unequivocally whether one mathematical expression is equivalent to another expression turns out to be very difficult. In fact, the problem has shown to be undecidable for various forms of expressions [13]. Nevertheless, in practice it is not hard to get a good idea if expressions are equivalent. We consider two expressions to be equivalent if they evaluate to the same result for any assignment of real numbers to their variables. This immediately suggests an approach to test the equivalence of expressions; we just choose assignments to the variables and check to see if the expressions agree for those assignments.

The main focus of this document will be to develop a variation of this randomized approach that utilizes a system of rounded interval mathematics. We will first outline two other variations of this approach that we investigated before deciding to focus on the interval method. The first of these is the floating point approach used in older versions of the Gateway System, and is highly instructive. The second is an approach using finite fields. Although it was quite mathematically interesting, this approach did not prove to be an effective solution to our practical problem. Both of these approaches will serve to illustrate some of the difficulties of the problem as well as to demonstrate the essence of the randomized approach. Before we go on to discuss the interval mathematics approach, we will present background material on floating point mathematics and interval mathematics. Most of this material, comprising Section 2.1 through Section 3.2 of this document, is background material that is original only in its presentation, not its content.

Once we have developed the basic system of rounded interval mathematics, we will work to apply interval mathematics methods to the problems of equivalence of expressions and equivalence of expressions up to an additive constant. The initial development of these approaches will lead to refinements both in our interval mathematical system and the equivalence-testing algorithms. In their final refined stages, these algorithms will represent the actual algorithms that we have implemented in the Gateway Web Testing System.

After we have shown the theoretical development of our algorithms, we will discuss the evaluation process that we have used to test the effectiveness of our implementation of these in real-world situations. We will discuss both correctness and speed of our implementation. Finally, we will talk about possible improvements to the algorithms and the conclusions of this thesis.

## 1.2 Careful Definition of Problem

Let  $\mathbb{R}$  represent the real numbers and  $\mathbb{R}^\dagger$  represent the real numbers with an added point called *NaN* (“Not a Number”). By an *expression* we mean a well-formed string of symbols which represents an  $n$ -place function from  $\mathbb{R}^n$  to  $\mathbb{R}^\dagger$ . Figure 1 contains a list of the primitive components we allow in expressions. 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*. Throughout this document we will use the phrases “the expression is undefined” and “the expression takes the value *NaN*” interchangeably.

We do not need to worry about details of syntax here; suffice it to note that the software and examples in this paper use the normal order of operations and allow juxtaposition to represent multiplication. An expression may contain numbers, constants, variables, operators, and functions, as well as parentheses and whitespace. For example,  $x$ ,  $\sin(x)$ , and  $-1*\sin(x/\pi)^2$  are all examples of expressions using just the variable  $x$ .

<b>Numbers:</b> 1234, 1234.5678, 1234.5678E9, etc.
<b>Constants:</b> e, pi
<b>Variables:</b> a, b, x, y, etc.
<b>Binary Operators:</b> +, -, *, /, ^
<b>Unary Negation Operator:</b> -
<b>Functions:</b> sin, cos, tan, log, ln, abs, sqrt, arcsin, arccos, arctan, asin, acos, atan, sec, csc, cot
<b>Delimiters:</b> (, ), space, tab

Figure 1: Primitives allowed in typical expressions.

The question we are concerned with is how to determine if two given 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  $\ln(\text{abs}(x))$ , on the other hand, do not represent the same function. For any negative  $x$  one of these evaluates to a real number and the other evaluates to  $NaN$ .

### 1.3 Simple Floating Point Approach

Previous versions of the gateway system have used a simple floating point approach to probabilistically determine the equivalence of two expressions. A comparison consists of a number of trials with different assignments of the variable. To carry out a trial, each variable present in the expressions is assigned a random floating point value in the range  $[-25, 25]$ . The expressions are then evaluated using floating point arithmetic. The values of the expressions are compared to see whether or not they agree.

For every assignment for which the expressions agree, we gain a little confidence that the expressions might indeed be equivalent. If, however, we find a single assignment where the expressions disagree, then the expressions cannot be equivalent<sup>1</sup>. Two expressions certainly cannot represent the same (well-defined) function if they produce different values for the same assignment!

The biggest hindrance to this scheme is the fact that floating point arithmetic is not exact. The implementation uses double precision floating point numbers; these use 64 bits of memory to represent each number. When one tries to represent all of the real numbers by a set of size  $2^{64}$ , something is necessarily lost. In fact, even when carrying out simple arithmetic on reasonably sized numbers, errors occur. For example, in Section 2.2 we will see that the equality  $0.1 + 0.2 = 0.3$  fails to hold in double precision floating point arithmetic. In a calculation which involves numerous elementary operations, we may be left with little idea of how accurate the final result is.

In order to avoid mistakes due to floating point errors, this initial algorithm (Algorithm 1) introduced a tolerance. Any values that agree within the tolerance were counted as equivalent. To compare the values of two expressions, we cannot just check if their values are equal. If the resulting values agree within 0.1 percent of their values then the trial is counted as a “hit.” If the values do not agree and are reasonably small, the trial is counted as a “miss.” If the values do not agree but are excessively large, the trial is counted as a “wide” and is ignored. This last category is introduced with the idea that processes which produce large floating point values are also likely to magnify the errors of floating point arithmetic, so the results are considered unreliable. The algorithm is shown in Figure 2.

In practice, it turns out that the 0.1 percent tolerance used was very large relative to the floating point errors in most computations. In fact, it was intentionally picked large to try to avoid errors where a correct answer is misgraded as incorrect. A larger difficulty was that such large tolerances allowed a fair number of blatantly incorrect answers to be graded as correct. There is another level to the difficulty, however. While it turns out that the algorithm was doing

<sup>1</sup>A term sometimes used for such a system of trials is *hypothesis testing*.

**Algorithm 1:** Probabilistic floating point check if expressions  $f$  and  $g$  are equivalent.

```
start with HITS and TRIALS equal to 0
repeat until TRIALS > MAXTRIALS
  assign random values to each variable in  $f$  and  $g$ 
  let  $u$  be the value of  $f$  calculated for those assignments
  and  $v$  be the value of  $g$  calculated for those assignments
  if  $|u - v| < \text{TOLERANCE} \times |u + v|$ 
    increment HITS
    if  $\text{HITS} \geq \text{HIT\_GOAL}$ 
      return TRUE (we found enough hits)
  else if  $|u| > \text{CEILING}$  and  $|v| > \text{CEILING}$ 
    (ignore this trial)
  else if  $u = \text{NaN}$  and  $v = \text{NaN}$ 
    (ignore this trial)
  else
    return FALSE (we found a miss)
  increment TRIALS
return FALSE (we failed to find enough hits)
```

Figure 2: Simple floating point algorithm for comparison of expressions  $f$  and  $g$ .

quite well in its grading, the method itself gives little assurance of that fact.

## 1.4 Finite Field Evaluation

The method of projection to a finite field that we examined is a variation of what is described by Gaston Gonnet in his two extended abstracts on determining equivalence of expressions in random polynomial time [2][3]. Gonnet makes extensive use of what he calls a “signature” function. Signature functions are best viewed as a parameterized class of functions that map from a subset of valid expressions to the integers modulo  $p$  for some fixed prime  $p$ . As such, they take the role of “hash functions,” assigning a specific hash code to each allowed expression. (In fact, Gonnet’s work builds on that of Martin [9], who did describe these as hash functions.)

Each signature function  $s$  must be such that if  $s(expr) \neq 0$  then  $expr \neq 0$ , and if  $expr \neq 0$  then there is a strong (preferably known) probability that  $s(expr) \neq 0$ . A signature function that has these properties allows us to create a test for the equality of expressions which has only one-sided error. To test if  $f = g$ , we simply test if  $s(f - g) = 0$  in a series of independent trials. If  $s(f - g) \neq 0$  in any trial, then we know conclusively that  $f \neq g$ . If, however,  $s(f - g) = 0$  for all trials, then we have a bound on the likelihood that  $f = g$ , though we do not have conclusive proof that the two are equal.

The signature function that Gonnet defines has as parameters the prime  $p$  and an integer value  $h(x)$  to assign to each variable  $x$  in the expression. It maps each integer  $n$  to  $n \bmod p$  and each variable  $x$  to  $h(x) \bmod p$ . The function is defined recursively, in such a way that  $s(a + b) = s(a) + s(b) \bmod p$ ,  $s(a - b) = s(a) - s(b) \bmod p$ ,  $s(a * b) = s(a) * s(b) \bmod p$ , and  $s(a/b) = s(a) * s(b)^{-1} \bmod p$ . The signature functions produced by (randomly) varying the parameters are assumed to be independent for purposes of having independent trials for testing if  $f = g$ .

There are several difficulties that wreak havoc with this approach. One basic problem comes in computing  $a^b$  if  $b$  is an arbitrary expression. It can be shown through an application of Fermat’s Little Theorem that  $b$  needs to be computed in modular arithmetic modulo  $p - 1$ . The integers modulo  $p - 1$  are not a field, so results are often undefined. In particular,  $b = 1/2$  causes great difficulty, so that square roots cannot always be computed. Another basic problem is that transcendental functions cannot be properly represented. The trigonometric, inverse trigonometric, exponential, and logarithmic functions can in some cases be simulated in modular arithmetic, but these simulations are troublesome. The ultimate result of these difficulties is that the class of expressions which can successfully be compared using the

method is too small to serve our needs.

## 2 Floating Point Arithmetic

### 2.1 Double Precision Floating Point Numbers

Since so much of the work that we do in this thesis depends on floating point arithmetic, it will be useful to have a good understanding of what floating point numbers actually are. For our purposes, the variety of floating point numbers that we are concerned with are double precision floating point numbers as implemented in the Java virtual machine. Since this is the only floating point format we are concerned with, we will use the phrases “floating point number” and “double precision floating point number” interchangeably. The canonical reference describing double precision floating point numbers is the standard produced by the IEEE in 1985 entitled *IEEE Standard for Binary Floating-Point Arithmetic* but more often referred to as IEEE-754 [6]. We have not had access to that document; the information in this section is based on secondary sources, especially [1].

The double precision floating point representation of real numbers uses 64 bits to specify a number in the approximate range  $[-1.7976931348623157 \times 10^{308}, 1.7976931348623157 \times 10^{308}]$  or one of the special values  $-\infty$ ,  $\infty$ , or *NaN*. The first bit, which we will number as bit 63, is the *sign bit*. The next eleven bits, bits 62–52, represent the *exponent* of the number. The last fifty-two bits, bits 51–0, represent the *mantissa* or *significand* of the number. Figure 3 illustrates this organization.

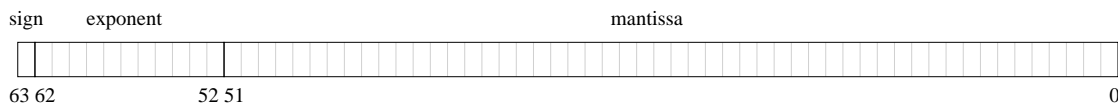


Figure 3: Double precision bit organization.

The following rules determine what number a floating point representation specifies. Let  $S$  be the integer represented by the sign bit (either 0 or 1),  $E$  be the positive integer represented by the exponent (from 0 to 2047), and  $M$  be the positive integer represented by the mantissa (from 0 to  $2^{52} - 1$ ). Let  $I = 1$  if  $E \neq 0$  and  $I = 0$  if  $E = 0$ . Let us write the binary rational notation  $I.M$  to mean the digit  $I$  followed by a binary point and the 52 digits of  $M$ . Thus  $I.M = I + 2^{-52}M$ . If  $E \neq 2047$  then the value represented by this floating point number is

$$(-1)^S I.M 2^{E-1022-I}.$$

If  $E = 2047$  and  $M = 0$  then the value is  $(-1)^S \infty$ . If  $E = 2047$  and  $M \neq 0$  then the value is *NaN*.

The reason that the above equation is a bit complicated is that there are two sorts of double precision floating point numbers: *normalized* and *denormalized* numbers. Floating point numbers are essentially written in the binary form of scientific notation. For proper scientific notation, the leading digit  $I$  should never be zero. Floating point numbers with  $I \neq 0$  are known as normalized numbers. Since the only nonzero binary digit is 1, there is no need to store this digit as part of the mantissa. This scheme does not allow us to produce normalized numbers with magnitude smaller than  $2^{-1022}$ , so to produce smaller numbers we have to allow the leading digit to be zero. Such numbers are known as denormalized numbers. Denormalized numbers fill in the gap between 0 and  $2^{-1022}$  using the same increments as normalized numbers have between  $2^{-1022}$  and  $2^{-1021}$ . The adjustment on the power of 2 from  $E - 1022$  to  $E - 1022 - I$  is necessary to correctly make the transition from normalized to denormalized numbers.

One very nice property that floating point numbers have, and which we take advantage of, is an easy-to-use ordering property. Disregarding the *NaN* values, the positive floating point numbers have the same lexicographic ordering as integers. The negative floating point numbers are in the reverse order of the integers. Thus, in order to get the next larger floating point number than a given (non-infinite) number  $n$ , we need only treat the binary representation of  $n$  as a 64 bit integer. If  $n$  is positive, we increment that integer, and if  $n$  is negative we decrement that integer. Conversely, in order to get the next smaller floating point number than a given (non-infinite) number  $n$ , we need only treat the binary representation of  $n$  as a 64 bit integer and do the reverse. If  $n$  is negative, we increment that integer, and if  $n$  is positive we decrement that integer.

## 2.2 Floating Point Arithmetic and Rounding

As we have already noted, it is obviously not possible to accurately represent all of the real numbers using only a finite subset of them. Errors are an inherent difficulty of floating point arithmetic. There are two classes of errors to be concerned with: *roundoff error* and *computational error*. By *roundoff error* we simply mean error that results from rounding a correct result to the next floating point number either higher or lower than that result. For example, the attempt to represent  $\pi$  as a floating point number inevitably results in roundoff error since  $\pi$  is not even a rational number, much less a floating point number. Even the attempt to represent 0.1 will result in roundoff error since 0.1 has a repeating binary representation. (This particular type of roundoff error, where a quantity which is exactly represented in one system is replaced by an inexact representation in another system, is known as *conversion error*.) A computation involving two floating point numbers may still suffer from roundoff error. For example, the attempt to divide 1 by 3 will not generate an exact result since the fraction  $1/3$  is not exactly representable as a floating point number.

By *computational error* we mean any error of a larger magnitude than roundoff error. If the result of a floating point computation is anything other than one of the two floating point numbers nearest the correct result, we say that a computational error has occurred. We will measure the magnitude of computational errors in terms of *Units in the Last Place (ULPs)*. The difference between a floating point number and the next larger or next smaller floating point number is one ULP. Note that the exact size of one ULP varies through the range of representable numbers; for the tiniest denormalized numbers one ULP is  $2^{-1074}$  while for the largest non-infinite numbers one ULP is  $2^{971}$ .

**Example 1** Let us examine why the sum  $0.1 + 0.2$  is not equal to 0.3 when carried out on a computer using double precision floating point arithmetic. The problem is not one of computational error, but rather an effect of multiple rounding errors. In ordinary usage involving anything besides whole numbers, nearly every step of every calculation is subject to rounding error.

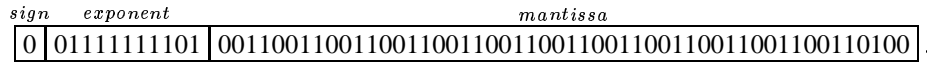
To carry out this calculation, we must first represent each of the decimal numbers 0.1, 0.2, and 0.3 as a floating point number. None of these numbers is exactly representable, so we convert each to the floating point number nearest to its value. The following assignments are therefore made:

$$\begin{aligned}
 0.1 &\mapsto \begin{array}{c} \text{sign} \quad \text{exponent} \qquad \qquad \qquad \text{mantissa} \\ \boxed{0 \mid 01111111011 \mid 1001100110011001100110011001100110011001100110011010} \\ = 0.0001100110011001100110011001100110011001100110011001100110011010 \quad (\text{base 2}) \\ = 0.100000000000000005511151231257827021181583404541015625 \quad (\text{base 10}) \end{array} \\
 0.2 &\mapsto \begin{array}{c} \text{sign} \quad \text{exponent} \qquad \qquad \qquad \text{mantissa} \\ \boxed{0 \mid 01111111100 \mid 1001100110011001100110011001100110011001100110011010} \\ = 0.0011001100110011001100110011001100110011001100110011010 \quad (\text{base 2}) \\ = 0.200000000000000011102230246251565404236316680908203125 \quad (\text{base 10}) \end{array} \\
 0.3 &\mapsto \begin{array}{c} \text{sign} \quad \text{exponent} \qquad \qquad \qquad \text{mantissa} \\ \boxed{0 \mid 01111111101 \mid 0011001100110011001100110011001100110011001100110011} \\ = 0.0100110011001100110011001100110011001100110011001100110011 \quad (\text{base 2}) \\ = 0.299999999999999988897769753748434595763683319091796875 \quad (\text{base 10}) \end{array}
 \end{aligned}$$

Once 0.1 and 0.2 have been represented as floating point numbers, we add the values of those representations. As it turns out, the result of this addition cannot be exactly represented as a double precision floating point number. The addition is carried out as follows, where the digits that represent the mantissa are underlined at each step.

$$\begin{aligned}
 &0.0001100110011001100110011001100110011001100110011001100110011010 \\
 + &0.00110011001100110011001100110011001100110011001100110011010 \\
 = &0.0100110011001100110011001100110011001100110011001100110011010 \\
 &\quad \text{which is rounded to} \\
 &0.010011001100110011001100110011001100110011001100110011010
 \end{aligned}$$

This rounded result is the floating point number



Observe that the representation for 0.3 differs by one ULP from the result of the addition of  $0.1 + 0.2$ . Note also that this difference is the result of rounding error only; there was no computational error involved. There was rounding error in representing each of the three numbers, and in representing the result of the addition. Such errors are unavoidable; there simply are not double precision floating point representations for every real number one encounters in everyday usage.

This example was chosen not to illustrate how serious floating point errors are, but rather how pervasive they are. Rounding errors happen on a very regular basis. In ordinary usage, it is often acceptable to simply assume that after a series of calculations the last few digits of a result are most likely garbage and are not to be trusted. It is difficult to know what portion of a result is garbage, however. One obvious problem is that if two similarly sized numbers are subtracted, and each suffered from a relatively small percentage of error, the result could be almost entirely error. Later operations, such as division by the error-fraught value, could further magnify that error. A classic example of this effect occurs when performing Gaussian elimination using floating point arithmetic.

For the most part, we will assume that we have access to machine implementations of arithmetic operations and elementary functions which suffer from rounding error but do not suffer from computational error. This is probably not a completely safe assumption. For a much-publicized example, the first-generation Intel Pentium chips produced computational errors in certain division operations. Less well-known is the fact that the implementations of  $\sin$  and  $\cos$  in the Intel 80387 math coprocessor routinely (about 28 percent of the time [8]) err by one ULP, and that chip's transcendental functions sometimes produce errors as large as 4.5 ULPs [7].

### 2.3 Java's Implementation of Floating Point Math

The implementations that we use for this project are in the Java language. This creates a disadvantage over programming in other languages. The Java language does not allow access to all of the features that the IEEE-754 standard requires compliant computer systems to implement. One feature which we would find useful is *directed rounding*. The IEEE specifications require that elementary arithmetic operations can be carried out in several different rounding modes. The default rounding mode, known as the "round to nearest" mode, is the only rounding mode that Java allows access to. In this mode, the results of a computation are always rounded to the nearest floating point number to the correct result. There are also "round towards infinity," "round towards negative infinity," and "round towards zero" rounding modes. In each of these modes, the result of a computation is rounded to the next floating point number in the specified direction. Another feature which we would find useful is the ability to determine whether or not the result of a computation has been rounded. The IEEE specifications provide a mechanism to do this; Java does not allow access to that mechanism.

Another concern is with Java's implementation of various mathematical functions. The basic arithmetic operations are undoubtedly carried out in hardware. Other than such examples as the Pentium division bug, these operations probably follow the IEEE standard, which does not allow these operations to produce any computational error. The accuracy of most other functions, such as the elementary trigonometric functions, is not specified by the IEEE standards. When those functions are implemented in hardware, the implementations vary widely from system to system [8].

The Java language specification seems to alleviate some fears about the problem of computational errors. The definitions of the numeric functions in Java's math library require that they produce the same results as the netlib package "Freely Distributable Math Library" (fdlibm) [14]. The Freely Distributable Math Library, in turn, claims to have error of less than one ULP for "major functions like  $\sin$ ,  $\cos$ ,  $\exp$ ,  $\log$ " [15]. This gives us some expectation that the math functions should be consistent across various hardware platforms and produce results free of computational error, at least for common functions.

In practice, actual implementations of the Java virtual machine do not seem to correspond to this specification. We have examined the results of Java's implementation of the  $\sin$  function on three different platforms: a Silicon Graphics machine running the IRIX operating system, a Pentium-based system running the Linux operating system,

and a Pentium-based system running the Windows 95 operating system. For large values of  $x$ , the values produced for  $\sin(x)$  varied widely among these systems. On the Silicon Graphics machine, whenever  $x \geq 2^{50}$ ,  $\sin(x)$  returns 0. On the Windows 95 machine, whenever<sup>2</sup>  $x \geq 2^{63}$ ,  $\sin(x)$  returns  $x$ . On the Linux machine,  $\sin(x)$  continues to return values distributed between  $-1$  and  $1$  for all finite values of  $x$ . Clearly, these implementations do not use the same algorithm to produce these varying results. At least two of these implementations certainly suffer from computational error. In short, we are left with little reason to trust the accuracy of Java math functions.

### 3 Interval Mathematics

#### 3.1 Exact Interval Mathematics

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 are bounds on the possible values of the result of the computation. The basics of the subject were laid out by Ramon Moore in 1966; this introduction quite closely follows his book [10].

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

**Definition 2** For  $A, B \in \mathcal{I}(\mathbb{R})$ , and for  $*$  one of the operations  $+, -, \times$ , or  $/$ ,

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

provided that  $a * b$  exists for all  $a$  in  $A$  and all  $b$  in  $B$ . (The provision fails only for division by an interval containing zero; we will take division by an interval containing zero to be undefined.)

Note that with the above definitions, each of the arithmetic operations produce intervals that are again in  $\mathcal{I}(\mathbb{R})$ . It is easy to make those mappings more explicit; suppose that  $A = [a_1, a_2]$  and  $B = [b_1, b_2]$ . Then

$$\begin{aligned} A + B &= [a_1 + b_1, a_2 + b_2] \\ A - B &= [a_1 - b_2, a_2 - b_1] \\ A \times B &= [\min\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}, \max\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}] \\ A / B &= [\min\{a_1/b_1, a_1/b_2, a_2/b_1, a_2/b_2\}, \max\{a_1/b_1, a_1/b_2, a_2/b_1, a_2/b_2\}] \end{aligned}$$

(assuming for the division that  $0 \notin B$ ).

The following example illustrates the effect of the above definitions.

#### Example 2

$$\begin{aligned} [-1, 1] + [3, 4] &= [2, 5] \\ [-1, 1] - [3, 4] &= [-5, -2] \\ [-1, 1] \times [3, 4] &= [-4, 4] \\ [-1, 1] / [3, 4] &= [-1/3, 1/3] \end{aligned}$$

Note what happens to an interval of width 0 in each of the definitions above. If the endpoints of  $A$  are identical and the endpoints of  $B$  are identical, the above definitions of arithmetic reduce to computation in  $\mathbb{R}$ . Thus we can consider  $\mathcal{I}(\mathbb{R})$  to be an extension of  $\mathbb{R}$ , where  $r \in \mathbb{R}$  is identified with the interval  $[r, r]$ .

For a continuous real-valued function  $f$ , there is a natural<sup>3</sup> interval extension of  $f$ .

<sup>2</sup>Actually this is not entirely true. For some values of  $x$ , sometimes  $\sin(x)$  returns  $x$  and sometimes  $\sin(x)$  returns  $NaN$ , depending apparently on what other mathematical operations have recently been performed. It is possible to let  $y = \sin(x)$ , do some other stuff, let  $z = \sin(x)$ , and not have  $y$  equal to  $z$ .

<sup>3</sup>It should be noted that this is not the unique interval extension of  $f$ . In general, any function defined on  $\mathcal{I}(\mathbb{R})$  which agrees with  $f$  on  $\mathbb{R}$  can be considered to be an interval extension of  $f$ .

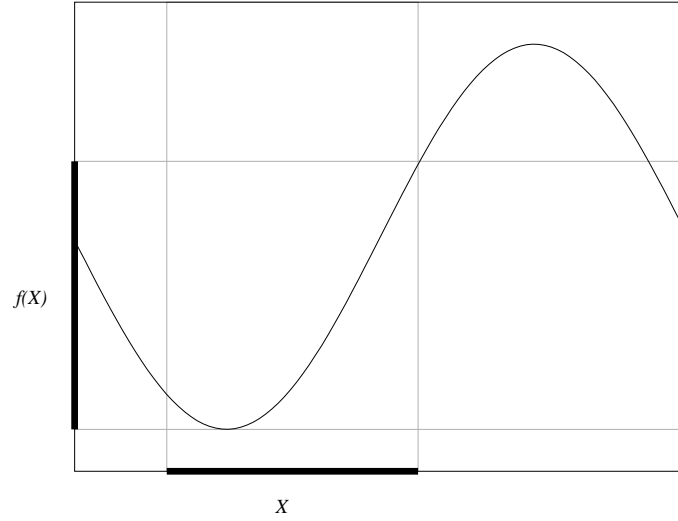


Figure 4: The natural interval extension of  $f(x) = \sin(x)$  applied to an interval  $X$ .

**Definition 3** For continuous functions  $f : \mathbb{R} \rightarrow \mathbb{R}$ , the *natural interval extension* of  $f$  is defined by  $f(A) = \{f(a) \mid a \in A\}$  for each  $A \in \mathcal{I}(\mathbb{R})$ .

Since the image of a closed and bounded interval under a continuous function is again a closed interval, this mapping does indeed map back into  $\mathcal{I}(\mathbb{R})$ .

Let's take a moment to examine some consequences of the system we have set up.

It is easy to observe that the intervals  $0 = [0, 0]$  and  $1 = [1, 1]$  serve as identity elements for addition and multiplication, respectively.

Addition and multiplication  $\mathcal{I}(\mathbb{R})$  are associative and commutative. These follow immediately from the associativity and commutativity of these operations on  $\mathbb{R}$ . Suppose that  $A, B, C \in \mathcal{I}(\mathbb{R})$  and  $*$  is one of the operations  $+$  or  $\times$ . Then

$$\begin{aligned} (A * B) * C &= \{(a * b) * c \mid a \in A, b \in B, c \in C\} \\ &= \{a * (b * c) \mid a \in A, b \in B, c \in C\} \\ &= A * (B * C) \end{aligned}$$

$$\begin{aligned} A * B &= \{a * b \mid a \in A, b \in B\} \\ &= \{b * a \mid a \in A, b \in B\} \\ &= B * A \end{aligned}$$

On the other hand, multiplication is not distributive across addition. Suppose again that  $A, B, C \in \mathcal{I}(\mathbb{R})$ .

$$\begin{aligned} A(B + C) &= \{a(b + c) \mid a \in A, b \in B, c \in C\} \\ &= \{ab + ac \mid a \in A, b \in B, c \in C\} \\ &\text{but} \\ AB + AC &= \{a_1b + a_2c \mid a_1, a_2 \in A, b \in B, c \in C\} \end{aligned}$$

What is happening here is somewhat deep. Whenever a term appears at more than one point in the computation, as  $A$  does in the second case above, the arithmetic does not treat these terms as identical. This is ingrained deeply in our construction of interval arithmetic. We think of an interval as representing a single real number whose exact value we

are unsure of. This mode of thinking can mislead us. When a term appears at more than one place in an expression, we have no assurance that that term represents the same value at each place in the calculation.

For exactly the same reason, in general there are not additive or multiplicative inverses. The obvious choice for an additive inverse to  $A$  would be  $-A$ , but the result of  $A - A$  is  $\{a_1 - a_2 \mid a_1, a_2 \in A\}$ . Similarly, the obvious choice for an additive inverse to  $A$  would be  $1/A$ , but the result of  $A/A$  is  $\{a_1/a_2 \mid a_1, a_2 \in A\}$ . So, for example,  $[1, 2] - [1, 2] = [-1, 1]$  and  $[1, 2]/[1, 2] = [1/2, 2]$ . The only intervals that have inverses are those which contain only a single value.

The proper mathematical categorization of  $\mathcal{I}(\mathbb{R})$  is therefore the monoid<sup>4</sup>, with operation of either addition or multiplication.

There is one further property of interval arithmetic which is essential to our purposes. That property is known variously as *inclusion monotonicity* [10] or *inclusion isotonicity* [4], and essentially guarantees that the result of a computation carried out on a larger interval will be a superset of the result of the computation carried out on a subinterval of that larger interval.

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

It is easy to see from the definition that the operations  $+$ ,  $-$ ,  $\times$ , and  $/$  are inclusion monotonic operations on  $\mathcal{I}(\mathbb{R})$ . Suppose that  $*$  is one of these operations and that  $A \subset A'$ ,  $B \subset B'$  are all real intervals. Then  $A * B = \{a * b \mid a \in A, b \in B\} \subset \{a * b \mid a \in A', b \in B'\} = A' * B'$ .

Our definition of the (natural) interval extension of real-valued functions also satisfies inclusion monotonicity. Suppose  $f : \mathbb{R} \rightarrow \mathbb{R}$  and  $A \subset A'$  are real intervals. Then  $f(A) = \{f(a) \mid a \in A\} \subset \{f(a) \mid a \in A'\} = f(A')$ .

## 3.2 Rounded Interval Mathematics

Our main interest in interval mathematics lies not in its properties as a mathematical system but for its uses as a computational tool. We will 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, we can develop an elegant way of calculating 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 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.

Let us outline the approach used to generalize interval mathematics from exact to rounded intervals. Just as in the previous section, this closely follows the method described by Moore [10]. Instead of considering intervals with endpoints of arbitrary real values, we will allow endpoints to have only those values which are representable in a specific computer implementation. This project is implemented entirely in Java, which provides double precision floating point numbers for its arithmetic. The following discussion will focus on that implementation; most of what we say would apply equally well to any other fixed set of machine numbers.

An *acceptable* rounded interval arithmetic scheme will be one in which the result of every operation is a machine interval which 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 the representations for positive and negative infinity are necessary to allow an acceptable rounded interval scheme to be implemented; without these there would be no way to enclose huge numbers in a machine interval.

**Definition 5** Let the set of *machine numbers*  $M$  be the set of double precision floating point numbers, excluding the floating point representations of *NaN*. The set of *machine intervals* is the set  $\mathcal{I}(M) = \{[x_1, x_2] \mid x_1, x_2 \in M\}$ .

---

<sup>4</sup>A *monoid* is simply a set endowed with a binary operation for which three properties hold: (1) the set is closed under the operation, (2) the operation is associative, and (3) there is an identity element for the operation.

Note that we still want to think of machine intervals as representing closed intervals on the real line. Although the endpoints are required to be machine numbers, the intervals still contain all the real values between the endpoints.

Let  $+_M$ ,  $-_M$ ,  $\times_M$ , and  $/_M$  be the binary operations defined on  $M \times M$  by the computer arithmetic operations as prescribed in the IEEE-754 specification in the “round to nearest” mode. These are operations from  $M \times M$  to  $M \cup \text{NaN}$ , where  $\text{NaN}$  occurs only as a result of division by zero. Note that these are *not* in general identical to their real-number arithmetic counterparts. (Recall the discussion of  $0.1 + 0.2 = 0.3$  in Section 2.2.) We will continue to use  $+$ ,  $-$ ,  $\times$ , and  $/$  when we mean binary operations from  $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ . If we use angled brackets to represent the operation of rounding to the nearest value in  $M$ , then the relationship between these is as follows. For  $*$  one of the operations  $+$ ,  $-$ ,  $\times$ , and  $/$  and for any  $a, b \in M$ ,  $a *_M b = \langle a * b \rangle$ .

For a quantity in  $M$ , let underlines and overlines represent the operations of subtracting and adding one ULP to that quantity, respectively. As we noted in Section 2.1, these operations can easily be implemented. One simply looks at the bit representation of a floating point number as a representation of an integer, decrements or increments that integer, and uses that new integer as a representation of the floating point number.

Define the following arithmetic on  $\mathcal{I}(M) \times \mathcal{I}(M) \rightarrow \mathcal{I}(M)$ . Suppose that  $A = [a_1, a_2]$  and  $B = [b_1, b_2]$  are both intervals in  $\mathcal{I}(M)$ . Define

$$\begin{aligned} A +_M B &= [\underline{a_1 +_M b_1}, \overline{a_2 +_M b_2}] \\ A -_M B &= [\underline{a_1 -_M b_2}, \overline{a_2 -_M b_1}] \\ A \times_M B &= [\min\{\underline{a_1 \times_M b_1}, \underline{a_1 \times_M b_2}, \underline{a_2 \times_M b_1}, \underline{a_2 \times_M b_2}\}, \max\{\overline{a_1 \times_M b_1}, \overline{a_1 \times_M b_2}, \overline{a_2 \times_M b_1}, \overline{a_2 \times_M b_2}\}] \\ A /_M B &= [\min\{\underline{a_1 /_M b_1}, \underline{a_1 /_M b_2}, \underline{a_2 /_M b_1}, \underline{a_2 /_M b_2}\}, \max\{\overline{a_1 /_M b_1}, \overline{a_1 /_M b_2}, \overline{a_2 /_M b_1}, \overline{a_2 /_M b_2}\}] \end{aligned}$$

(assuming for the division that  $0 \notin B$ ).

This defines an acceptable, but not optimal, scheme of rounded interval arithmetic. It is acceptable because as long as the machine operations  $+_M$ ,  $-_M$ ,  $\times_M$ , and  $/_M$  produce results that are within one ULP of the correct real number result, the underline and overline operations will expand the machine interval to contain that correct result. It is not optimal because somewhere around fifty percent of the time the machine arithmetic operation on a given endpoint will have already rounded its result in the proper direction, in which case the underline or overline operation is unneeded and expands the interval unnecessarily. Note also that if endpoints lie at plus or minus infinity, the machine arithmetic operations are defined to do the correct things, so the machine interval operations will also do the correct things.

It takes a little more thought to make the natural extension of real-valued functions carry through to an acceptable scheme of rounded interval mathematics. In particular, one has to be very careful with functions which are not monotone over the interval with which one is concerned. We will not detail what needs to be done for every function we allow; we will explain the general case and give two examples to illustrate that such schemes can be implemented in a practical way.

Let us begin with an easy example. The natural logarithm function is simple to extend into an inclusion monotonic rounded form since it is monotonic throughout its domain of definition. Let  $\ln_M : M \rightarrow M$  be a machine implementation of the natural logarithm function that is free from computational error. That is to say,  $\ln_M(x) = \langle \ln(x) \rangle$ . Suppose that  $A = [a_1, a_2]$  is an interval in  $\mathcal{I}(M)$  and that  $A$  does not contain any negative numbers. Define

$$\ln_M(A) = [\underline{\ln_M(a_1)}, \overline{\ln_M(a_2)}]$$

Note that by the assumption that  $\ln_M$  is free from computational error we know that  $\underline{\ln_M(a_1)}$  is a lower bound for  $\ln(a_1)$  and  $\overline{\ln_M(a_2)}$  is an upper bound for  $\ln(a_2)$ . Then, since the  $\ln$  function is monotonically increasing, this guarantees that the interval  $[\underline{\ln_M(a_1)}, \overline{\ln_M(a_2)}]$  contains all values  $\ln(a)$  for  $a_1 \leq a \leq a_2$ . Thus this defines an acceptable rounded interval version of the  $\ln$  function.

The general principal 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 values on the function’s value. It is important to recall that we have no desire to hand-create rounded interval versions of every function  $f : \mathbb{R} \rightarrow \mathbb{R}$ ; we just need to create rounded interval versions of the elementary functions. (The functions we care about are shown in Figure 1.) None of these functions has terribly wild behavior. For each of these functions, given an interval of finite size, we can divide that interval into a finite set of subintervals on which the function is

monotonic. Then we only have to consider the function’s value on the endpoints of those subintervals. For an infinite interval, we just need some bound on the function’s value. If need be, we can take  $[-\infty, \infty]$  as that bound and still satisfy the requirements of acceptability.

The example of the  $\sin$  function will make this more clear. Suppose that  $\sin_M : M \rightarrow M$  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  $\overline{\sin_M(a_1)}$  and  $\overline{\sin_M(a_2)}$ .

One of course has to be a bit careful in checking whether  $\sin$  might have extrema on  $A$ . Recall that  $\sin$  has local maxima at  $2n\pi + \pi/2$  for integers  $n$  and has local minima at  $2n\pi + 3\pi/2$  for integers  $n$ . One quick check is that if the interval width is greater than  $\frac{2\pi}{M}$ , we can assume that  $\sin$  will have at least one maximum point and one minimum point on the interval. So if  $\overline{a_2 -_M a_1} > \frac{2\pi}{M}$  then  $\sin_M(A) = [-1, 1]$ . To check for extrema for narrower intervals, one can divide  $A$  by the interval  $[\frac{2\pi}{M}, \frac{2\pi}{M}]$  and check for the inclusion of points  $n + 1/4$  and  $n + 3/4$  for integers  $n$ . This method ensures that whenever  $A$  contains an extremum that extremum is reported. Sometimes this method will report extrema which are actually outside of  $A$ , but this just results in overestimation of the resulting interval and inclusion monotonicity is maintained.

### 3.3 Interval Solution of the Expression Equivalence Problem

**Algorithm 2A:** Probabilistic rounded interval check if expressions  $f$  and  $g$  are equivalent.

```

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  $U \cap V = \emptyset$ 
    return FALSE (the functions cannot be equal)
  increment TRIALS
return TRUE (if cannot demonstrate that  $f$  and  $g$  differ, assume they are equal)

```

Figure 5: The initial version of a rounded-interval algorithm for comparison of expressions  $f$  and  $g$ .

The procedure to test if two expressions  $f$  and  $g$  are equivalent using interval arithmetic is much the same as when using floating point arithmetic or evaluation in a finite field. We perform 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 enough trials in which the intervals overlap, the expressions are judged to be equal. This algorithm is shown in Figure 5, and an illustration of the interval comparison is shown in Figure 6.

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 real value of  $f$  at that point. The same thing 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 can be restricted to have only one-sided error<sup>5</sup>. While unequal

<sup>5</sup>We will actually violate this property intentionally in order to avoid the possibility of “skeleton key” functions which produce such wide intervals at each point as to be judged equal to any other function. This is discussed in section 4.3.

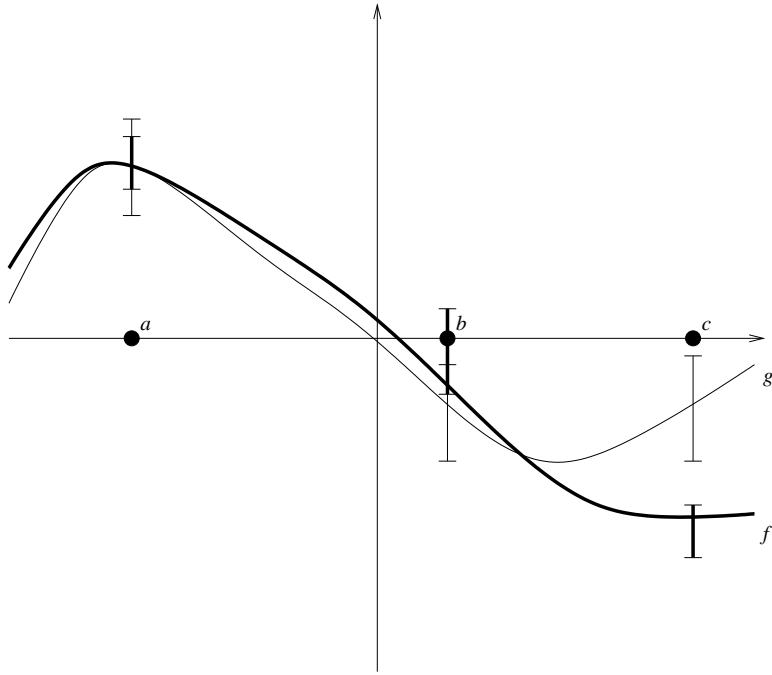


Figure 6: Interval comparison of functions. The curves represent the actual values of  $f(x)$  and  $g(x)$ ; the error bars show interval evaluations of  $f$  and  $g$  at the values  $a$ ,  $b$ , and  $c$ . Note that the intervals overlap at points  $a$  and  $b$ , so the functions are compatible at those points. At point  $c$ , however, the intervals do not overlap so that sample shows the functions are not equal.

expressions may sometimes be close enough in value to be judged correct, equivalent expressions should never be mistakenly judged to be unequal.

There is one important point that we have glossed over thus far in our discussion. In Section 1.2 where we made a careful definition of the problem, we said that we were concerned with functions from  $\mathbb{R}$  to the set  $\mathbb{R}^\dagger = \mathbb{R} \cup \{NaN\}$ . The reason for this slightly unorthodox definition is to treat functions with varying domains of definition more uniformly. The functions  $\ln(x)$  and  $\ln(|x|)$  are not equivalent; the first produces a result of *NaN* for negative  $x$ , while the second produces a result that lies in  $\mathbb{R}$ . So far we have considered only the case where the result of an expression is a real number, and have ignored the possibility that the expression evaluates to *NaN* on a given trial.

There are three different cases to consider when we allow expressions to take on the value *NaN*. Let us first consider the operation of division. When we defined the real interval operation of division, we left  $A/_M B$  undefined in the case where  $0 \in B$ . 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 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  $A/_M B = \textit{Certainly NaN}$ . The third case is that  $0 \in B$ , but  $B$  also contains nonzero values. Now, there are some combinations of  $a \in A$  and  $b \in B$  such that  $a/b \in \mathbb{R}$ . We will describe this as  $A/_M B = \textit{Possibly NaN}$ . This allows us to define the division operation  $A/_M B$  as an operation which is defined for all  $A, B \in \mathcal{I}(M)$  and which maps from  $\mathcal{I}(M) \times \mathcal{I}(M)$  to  $\mathcal{I}(M) \cup \{\textit{Certainly NaN}\} \cup \{\textit{Possibly NaN}\}$ .

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  $\mathbb{R}$  to  $\mathbb{R}^\dagger$ . For any real  $x \geq 0$  we have  $\sqrt{x} \in \mathbb{R}$ . 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:  $\sqrt{A} \in \mathcal{I}(M)$ . If  $A$  contains only negative values we have case two:  $\sqrt{A} = \textit{Certainly NaN}$ . If  $A$  contains both negative and nonnegative values we have case three:  $\sqrt{A} = \textit{Possibly NaN}$ <sup>6</sup>.

**Algorithm 2B:** Probabilistic rounded interval check if expressions  $f$  and  $g$  are equivalent.

```

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  $U = \textit{Certainly NaN}$  and  $V \in \mathcal{I}(M)$ 
    return FALSE (we have found a miss)
  else if  $U \in \mathcal{I}(M)$  and  $V = \textit{Certainly NaN}$ 
    return FALSE (we have found a miss)
  increment TRIALS
return TRUE (if cannot demonstrate that  $f$  and  $g$  differ, assume they are equal)

```

Figure 7: Second version of a rounded-interval algorithm for comparison of expressions  $f$  and  $g$ .

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 7.

<sup>6</sup>A slightly different approach than the trichotomy we presented would be to extend to the system to map into  $\mathcal{I}(M) \cup \{NaN\} \cup \{[a, b] \cup \{NaN\} \mid a, b \in M\}$ . The three sets in this union correspond to the three cases we used here. The third set corresponds to the case we lumped together as *Possibly NaN*, but provides more detail as to what other values besides *NaN* are possible.

### 3.4 Interval Solution of Expression Equivalence Modulo a Constant

In the grading of calculus exams, it is often useful to be able to determine if one expression is equivalent to another expression up to some fixed constant. That is, given two expressions  $f$  and  $g$ , we want to determine if there exists a constant  $C$  such that  $f = g + C$ . This question arises when students are asked to compute the indefinite integral of a given expression. Different methods of integration of the same expression can produce results that 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 expressions. 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. This algorithm is shown in Figure 8, and an illustration of the interval comparison is shown in Figure 9.

This process shares the nice property of having only one-sided error. Two expressions which are equivalent modulo a constant should never be judged to be not equivalent. To see this, suppose that  $f$  and  $g$  are expressions such that  $f = g + C$  for some constant  $C \in \mathbb{R}$ . 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.

**Algorithm 3A:** Probabilistic rounded interval check if expressions  $f$  and  $g$  differ by an additive constant.

```

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  $\text{INTERSECTION} \cap (U -_M V)$ 
    if INTERSECTION =  $\emptyset$ 
      return FALSE (we have found a miss)
    else if  $U = \text{Certainly NaN}$  and  $V \in \mathcal{I}(M)$ 
      return FALSE (we have found a miss)
    else if  $U \in \mathcal{I}(M)$  and  $V = \text{Certainly NaN}$ 
      return FALSE (we have found a miss)
    else
      (do nothing; the trial is a wide)
  increment TRIALS
return TRUE (there is still a range of constants by which  $f$  and  $g$  might differ)

```

Figure 8: A rounded-interval algorithm for comparison of expressions  $f$  and  $g$  up to an additive constant.

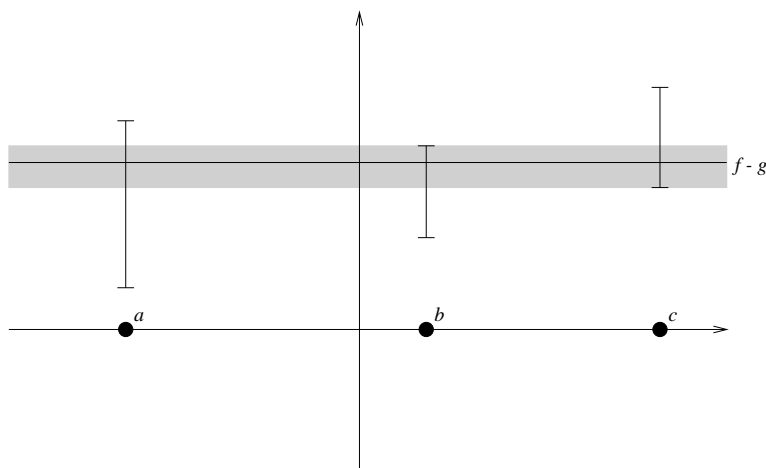


Figure 9: Interval comparison of functions modulo a constant. Intervals are evaluated for  $f - g$  at sample points  $a$ ,  $b$ , and  $c$ . The shaded area represents the intersection of these intervals. As long as that intersection is non-empty, it is possible that  $f$  and  $g$  differ by a constant.

## 4 Implementation

### 4.1 The Gateway System

After all of the details of the preceding discussion, it is time to step back a bit and recall what the big picture is. The ultimate goal of this is to implement and evaluate an algorithm to grade free responses to mathematical questions. This implementation lives inside of the Gateway Web Testing System, which takes care of generating and serving tests, of recording scores, and the carrying out the rest of the myriad of things that need to occur in an online testing system. All of this structure has been in place, running, and thoroughly tested over the past several semesters.

Our goal in implementation has therefore been to seamlessly integrate interval evaluation routines into the existing gateway system. The gateway system is divided into a number of packages<sup>7</sup>, each of which carries out different functions. Almost all of the changes needed to implement interval evaluation routines into the system came in the *parser* package, which is so named because it is the home of classes that parse the text of a mathematical expression into a binary tree suitable for machine evaluation. Classes in the parser package do not just parse expressions; they are also responsible for evaluating expressions mathematically, comparing different expressions, and producing formatted representations of expressions. A few changes also needed to be made in the *question* package, which is so named because it is the home of classes that represent templates of different question styles in an exam.

### 4.2 The `ia_math` Package

We made use of a publicly available implementation of interval mathematics for the Java platform that could be adapted to meet our needs. This implementation is a Gnu Public Licensed<sup>8</sup> library of Java routines known as the *ia\_math* package [5]. The *ia\_math* package included Java classes to represent machine intervals and provided the functionality to perform rounded arithmetic on the intervals, to apply most elementary functions to the intervals, and to do other useful things such as intersect intervals.

<sup>7</sup>A *package* in Java programming is a collection of classes, usually related, that reside in the same directory on the drive. A *class* in Java is a program or subprogram that resides in its own file in that directory. A single class may represent something as tiny as a single numerical value or as large as an entire application.

<sup>8</sup>The Gnu Public License (GPL) requires the library to be distributed with full source code included, and allows modification to that source code. The modified library continues to be covered by the GPL. It is permissible, however, for a commercial program to make use of the library routines without itself being included under the GPL. In our case, we make use of a modified copy of the *ia\_math* package. If this improved system is commercially distributed, then, it will include the full source code of the modified *ia\_math* library, which will remain under the GPL. The remainder of the software package, which is not derived from the *ia\_math* library, does not fall under the GPL.

The `ia_math` package makes a compromise between speed and accuracy. This compromise is largely embodied in the fact that the `ia_math` package assumes that the Java implementations of basic math functions are free of computational error. As we noted in section 2.3, this assumption is almost certainly incorrect. All of the `ia_math` implementations of the natural interval extension of real-valued functions are based on the existing real-valued implementations of those functions, so if and when computational errors occur, the interval results could lose the property of inclusion monotonicity. In other words, one would not want to use this package for air traffic control or design of a nuclear facility<sup>9</sup>. In practice, such errors do not seem to be debilitating. One counterbalance is that the interval bounds of a result are rounded outwards after every operation whether they need it or not. Thus, in a long calculation, a buffer zone of excess rounding tends to accrue. This excess rounding means that occasional computational errors of small magnitude might still be covered by the buffer.

Not everything in the original `ia_math` package met our needs perfectly. There were a few minor oversights; for example the package did not include an interval implementation of the absolute value function. The one major way in which the `ia_math` package failed to meet our needs was in its handling of points where functions do not exist. In this thesis we have developed the distinction between the interval extension of a real valued function producing the results *Certainly NaN* and *Possibly NaN*. The `ia_math` package made no such distinction. The original behavior was to throw an exception<sup>10</sup> only in cases where there the result was undefined for the entire interval. When a function applied to an interval would produce *NaN* for part of the interval and real values for the remainder, the `ia_math` routines would return an interval representing the possible range of real values that the function would produce. For example,  $\sqrt{[-1, 4]}$  would return the interval  $[0, 2]$ .

That behavior is undesirable for our purposes. Consider the following (admittedly somewhat contrived) example. Suppose  $f$  is the expression  $10^{(-200)} * \text{sqrt}(x)$  and  $g$  is the expression  $\text{sqrt}(10^{(-400)} * x)$ . Note that these two expressions are completely equivalent. Suppose that  $x$  is picked to be a reasonably sized negative number, say  $x = -10$ . When  $f$  is evaluated using the unmodified `ia_math` package, it will throw an exception because the square root of  $-10$  cannot be evaluated. On the other hand, when  $g$  is evaluated, the product inside the square root will be smaller than any representable negative number and will be rounded up to include or even extend past zero. In the original `ia_math` scheme, the square root of such an interval would be zero or maybe even a tiny positive machine interval. Thus two equivalent expressions could have sample points where one expression failed to exist and the other expression did claim to exist. Furthermore, despite the rather contrived nature of our example, this situation has been observed to occur in practice comparing a student's response to the correct answer on a calculus question.

One possible work-around to the above difficulty would be to treat trials in which one expression exists and the other does not as wides. This work-around is also highly undesirable. It is easy to construct expressions that exist nowhere (e.g.  $\text{sqrt}(-1)$ ). Students—presumably inadvertently—do it all the time on calculus exams. (A favorite real-world example is the expression

$$\left( \left( 3(2x-3) - 2(3x-4) \right) / \left( (2x-3)^2 \right) \right)^{-1/2}$$

which was included in several student responses to a tricky differentiation problem. It is left as an exercise to the reader to check that this indeed fails to exist as a real number for any value of  $x$ .) An expression that exists nowhere would never generate a miss in comparison with any other expression!

The introduction of the two cases *Certainly NaN* and *Possibly NaN* is a much more effective solution to these difficulties. In typical situations where an expression fails to exist, it does so in a drastic way; for instance an expression will take the square root or natural logarithm of an interval containing only negative numbers. Such cases produce the *Certainly NaN* condition. When compared to another expression that does exist at that point, the trial will conclusively show that the expressions are not equivalent. Typical expressions only produce *Possibly NaN* in boundary conditions between variable assignments that produce *Certainly NaN* and an interval result. Although a trial that produces *Possibly NaN* cannot produce a miss, this is not a problem if such trials occur relatively infrequently in comparison to more useful trials.

There is one situation in which the trichotomy of intervals, *Certainly NaN*, and *Possibly NaN* produces unsatisfying results. That situation is the application of the power operation to raise a negative number to some power. We have

<sup>9</sup>This is a tongue-in-cheek comment in reference to a similar disclaimer that is included in the source code of every example program that Sun distributes with the Java Development Kit.

<sup>10</sup>In Java programming, when a routine *throws an exception*, it gives up on doing whatever it was attempting to do and continues on at the place where the exception is *caught*.

not discussed the power operation before, but it is defined just as other operations. The interval version of  $\hat{\phantom{x}}$  is defined, for  $A = [a_1, a_2] \in \mathcal{I}(\mathbb{R})$  and  $B = [b_1, b_2] \in \mathcal{I}(\mathbb{R})$ , as

$$A \hat{\phantom{x}} B = \{a^b \mid a \in A, b \in B\}$$

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

We will take the function  $a^b$  to be defined as the principal branch of  $a^b$  evaluated as a complex number. With this definition, when  $a$  is negative the quantity  $a^b$  exists if and only if  $b$  is an integer. One nice property of this scheme is that the expression  $x^{(1/2)}$  is treated identically to expressions such as  $\text{sqrt}(x)$ . We do not provide a cube root function that exists on the negative numbers, so there is no danger of confusion between taking the cube root of a value and raising it to the  $1/3$  power.

When  $A$  contains only positive values, the power function is continuous with respect to each argument and so the result of  $A \hat{\phantom{x}} B$  is again a closed interval. When  $A$  contains negative values, the operation  $A \hat{\phantom{x}} B$  is defined if and only if  $B$  contains only a single value  $b \in \mathbb{Z}$ . Then for that fixed value  $b$ , the function  $a^b$  is continuous with respect to  $a$ , so the result of  $A \hat{\phantom{x}} B$  is again a closed interval.

Consider now the rounded interval version of this operation. Suppose  $A = [a_1, a_2]$  is a machine interval that contains negative values and  $B = [b_1, b_2]$  is another machine interval. If  $B$  contains no integer values, then  $A \hat{\mathcal{M}} B = \text{Certainly NaN}$ . If  $b_1 = b_2 = n$  for some integer  $n$ , then  $A \hat{\mathcal{M}} B$  results in a machine interval. In the remaining case,  $B$  contains at least one integer value but also contains non-integer values. Thus  $A \hat{\mathcal{M}} B$  results in *Possibly NaN*.

The unsatisfying feature of all of this is that the expressions  $x^2$  and  $x^{(1+1)}$  are not treated in quite the same way. Since we do not trust the result of any machine operation, when we add the interval  $[1, 1]$  to itself the result is the interval  $[\underline{2}, \overline{2}]$ , which has nonzero width. Therefore, for negative values of  $x$  the expression  $x^2$  evaluates to an interval while the expression  $x^{(1+1)}$  evaluates to *Possibly NaN*. Thus the result of any trial in which  $x$  is chosen to be negative will be a wide. More seriously, the expression  $(-1)^{(1+1)}$  will always evaluate to *Possibly NaN*, so regardless of what expression to which it is compared, the result of each trial will be wide.

### 4.3 Implementation of Expression Equivalence Algorithm

There are a number of details that our description of the interval mathematical algorithm for determining equivalence of expressions left unspecified. We discussed a series of trials and what possibilities could occur in a single trial. We did not, however, specify how the random values would be chosen for each trial. Nor did we specify what would constitute a sufficient regimen of trials to judge a pair of expressions to be equivalent. We alluded in a footnote to the idea of skeleton key functions, which would lead to another violation of the one-sided error principles. In this section we will clear up each of those details.

Recall that in order to determine if two expressions  $f$  and  $g$  are equivalent, we were to carry out a series of trials. In order for a trial to be useful, we would like there to be a good chance that if  $f$  and  $g$  are not equivalent expressions then that trial will produce a miss. With this goal in mind, we can evaluate the effectiveness of different methods of choosing the random values to assign to variables in each expression. Our choices are made for the sake of effectiveness in practice, and hence are dependent on what sorts of expressions we will see in practice. One key property of expressions that we see in our situation is that the expressions tend to have reasonably sized results when the variable assignments are reasonably small. Many expressions that occur in practice grow large quickly as the variable assignments grow larger in magnitude. The domain of definition of most expressions that we see in practice tends to include small numbers.

These properties have intuitive support and also empirical backing; as we will describe in Section 5.3, we have been able to look at every student response to every calculus question asked last semester in order to empirically determine what decisions are most effective. In the case of assigning random values to the variables in expressions, we have found that a Gaussian distribution produces a more effective set of sample points than the uniform distribution that was previously in use. The center of the distribution that we use is quite naturally placed at 0, and one standard deviation is placed at 10.0. These values are chosen to work well with the questions that are in the question database; it would be easy to come up with questions for which these values produce very poor results.

For each individual trial, we have already discussed the possibility that the trial produces a miss. Recall that a *miss* is any condition which ensures that the two expressions cannot be equivalent. In order to decide how many trials are sufficient to declare two expressions to be equivalent, we will discriminate more carefully between different sorts

of trials that do not result in a miss. If both expressions evaluate to machine intervals, and those intervals overlap, this serves to increase our confidence that the expressions are equivalent. If, however, one expression evaluates to *Possibly NaN*, then this does much less to bolster our confidence that the expressions are equivalent. In general, we will categorize trials that increase our confidence that the expressions are equivalent as *hits*, while we will say that trials are *wide* when they do little to increase our confidence that the expressions are equivalent.

We identify one remaining case in which a trial might do little to increase our confidence that a pair of expressions are equivalent. Suppose that a clever user had some idea how our evaluation scheme works. That user could come up with a *skeleton key function* that was designed in such a way as to produce enormously wide intervals whenever it was evaluated. A possible example would be  $10^{300} * \sin(10^{100})$ . Since  $10^{100}$  is so large, the rounding will produce an interval of width greater than  $2\pi$ . Then when the sin function is applied to this interval, the result will be  $[-1, 1]$ . Multiply that interval by a huge number, and the result closely resembles  $[-\infty, \infty]$ . Without anything to prevent such abuses, a skeleton key function would produce intervals that overlap the intervals produced by just about any other expression. This would allow an easy match to any function which is defined everywhere. Although it seems unlikely that anyone would try to or succeed in producing such an expression, if someone did it would provide an easy way to cheat the system.

In order to prevent skeleton key functions, we introduce a check to ensure that the user expression does not produce intervals that are too wide. This produces an asymmetry between our treatment of the user and correct expressions. If the interval produced by the user expression is too wide in comparison to the interval produced by the correct expression, the trial is declared to be a wide. The reason for comparing the width of the user expression to that of the correct expression is to ensure that if the user writes an equivalent expression in similar form to that of the correct expression, then the user's expression should not be counted wide. We need the asymmetry since although the user's expression is not allowed to produce intervals much wider than those of the correct expression, it is perfectly fine for the user's expression to produce intervals which are much narrower than those of the correct expression. Furthermore, the essential measure of the size of an interval is its width in ULPs. The size of one ULP depends on the value of the expression at that point. Thus the check to see if the user interval is too wide should depend both on the size and value of the correct expression's interval in that trial.

Now that we have distinguished carefully between trials that are hits, misses, or wides, we can carefully describe how many trials we consider sufficient to decide that a pair of expressions are equivalent. We will revise our former algorithm once more to do this; this revision is presented in Figure 10 as Algorithm 2C. There are four different termination conditions for the series of trials. The first, which we have already discussed, is that we can stop the trials whenever a single miss is found. In this case we can immediately declare the functions to be unequal.

The second termination condition occurs when a threshold number of hits is reached. That threshold is set at 14 for the empirical reason that that is about twice the number of hits that any non-matching pair of expressions received in a large set of real-world data. This will be discussed in greater detail in Section 5.3.

The third termination condition occurs when a threshold number of trials is reached in which the correct expression evaluated to an interval. (Let us call a trial in which the correct expression evaluates to an interval a *sample*.) The thought is that, if the user expression and correct expression are equivalent, almost all of the time when one returns an interval the other should as well. The difficulty is that, as we observed, a pair of equivalent expressions can produce trials in which one expression produces an interval and the other expression produces *Possibly NaN*. Therefore this sample threshold should be higher than the hit threshold to allow for bad luck finding *Possibly NaN* results of the user expression. The value we use is set rather arbitrarily at 100. If this threshold is reached, the process terminates and returns a judgment that the expressions are not equivalent.

The fourth and final termination condition occurs when a huge number (100,000) of trials have been carried out and none of the other termination conditions have been reached. This should never occur, since there should not be questions in the question database that fail to produce intervals for a significant portion of randomly assigned values. This fourth condition is essentially an escape clause to make sure the program does not get stuck if something goes wrong. Again a judgment is returned that the expressions are not equivalent.

The fact that the third and fourth termination conditions judge the expressions to be unequal means that we no longer have just one-sided error. A correct expression could be numerically unstable enough to be measured wide all of the time by the test that we implemented to stop skeleton key functions. It might also be measured wide all of the time by somehow managing to produce the result *Possibly NaN* for each trial. Or a bad question might get put into the question database. Since it seems so unlikely that a correct expression would fall through to these termination

**Algorithm 2C:** Probabilistic rounded interval check if the user expression  $f$  and the correct expression  $g$  are equivalent.

```
start with TRIALS, SAMPLES, and HITS equal to 0
repeat until TRIALS > MAXTRIALS or SAMPLES > MAXSAMPLES
  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  $V \in \mathcal{I}(M)$ 
    if  $U \cap V = \emptyset$ 
      return FALSE (we have found a miss)
    else if the width of  $U$  is not large compared to the width and upper endpoint of  $V$ 
      increment HITS
      if HITS > HIT_GOAL
        return TRUE (the expressions are a good match)
    else
      (do nothing; the trial is a wide)
  else if  $U = \textit{Certainly NaN}$  and  $V \in \mathcal{I}(M)$ 
    return FALSE (we have found a miss)
  else if  $U \in \mathcal{I}(M)$  and  $V = \textit{Certainly NaN}$ 
    return FALSE (we have found a miss)
  else
    (do nothing; the trial is a wide)
  increment TRIALS
  if  $V \in \mathcal{I}(M)$ 
    increment SAMPLES
return QUALIFIED FALSE (we think the expressions are unequal but do not know this)
```

Figure 10: Final version of a rounded-interval algorithm for comparison of expressions  $f$  and  $g$ . This version takes into account *Certainly NaN* and *Possibly NaN* conditions as well as skeleton key functions.

conditions, the judgment seems warranted. In recognition of the fact that the judgment is uncertain, however, we have rigged a special message to appear by the graded response when the test results are shown to the user. That message declares, “The comparison routine was unable to confidently grade this response.” This message will probably not often be seen; neither the third nor fourth thresholds were ever approached in testing on recorded student responses from last semester’s classes.

#### 4.4 Implementation of Variations

In Section 3.4 we described an algorithm (Algorithm 3A) for testing equivalence of expressions up to an additive constant. Algorithm 3A was very similar to Algorithm 2B; the only difference was in what conditions produce a miss when both expressions evaluate to an interval. Suppose that  $f$  and  $g$  are the expressions in question, and they evaluate to the machine intervals  $U$  and  $V$ , respectively, for a particular assignment to the variables. In each of the Algorithm 2 variations, a miss is produced if  $U \cap V = \emptyset$ . In Algorithm 3A, a miss is produced if  $\text{INTERSECTION} \cap (U -_M V) = \emptyset$ , where  $\text{INTERSECTION}$  is an interval which keeps a running intersection of the differences  $U -_M V$  for each trial. Since the algorithms are so similar, the implementations can also be similar. Figure 11 shows the final version of the algorithm (Algorithm 3B), which is exactly analogous to Algorithm 2C.

**Algorithm 3B:** Probabilistic rounded interval check if expressions  $f$  and  $g$  differ by an additive constant.

```

start with TRIALS, SAMPLES, and HITS equal to 0 and INTERSECTION equal to  $[-\infty, \infty]$ 
repeat until TRIALS > MAXTRIALS or SAMPLES > MAXSAMPLES
  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  $\text{INTERSECTION} \cap (U -_M V)$ 
    if INTERSECTION =  $\emptyset$ 
      return FALSE (we have found a miss)
    else if the width of  $U$  is not large compared to the width and upper endpoint of  $V$ 
      increment HITS
      if HITS > HIT_GOAL
        return TRUE (the expressions are a good match)
    else
      (do nothing; the trial is a wide)
  else if  $U = \text{Certainly NaN}$  and  $V \in \mathcal{I}(M)$ 
    return FALSE (we have found a miss)
  else if  $U \in \mathcal{I}(M)$  and  $V = \text{Certainly NaN}$ 
    return FALSE (we have found a miss)
  else
    (do nothing; the trial is a wide)
  increment TRIALS
  if  $V \in \mathcal{I}(M)$ 
    increment SAMPLES
return QUALIFIED FALSE (we think the expressions are unequal but do not know this)

```

Figure 11: Final version of a rounded-interval algorithm for comparison of expressions  $f$  and  $g$  up to an additive constant. This version takes into account *Certainly NaN* and *Possibly NaN* conditions as well as skeleton key functions.

The Gateway Web Testing System also provides two other question modes that require simple extensions of the algorithms we have described. In the first of these question modes, the user is asked to enter a vector, where each component of the vector can be an expression. The user is allowed to use vector addition and scalar multiplication

in order to construct his or her response. The user response vector is correct if every component of that vector is an equivalent expression to the corresponding component of the correct answer vector. We implement this question mode by reducing this to a repeated application Algorithm 2C. Any vector addition or scalar multiplication in the user response and in the correct answer is carried out to produce a vector of expressions. For example, the expression  $2[3x, 3y, 0] + [1, 1, z]$  is translated into the the vector of expressions  $[2*3x+1, 2*3y+1, 2*0+z]$ . Once the expressions are in this form, we apply Algorithm 2C to each component of the vectors in turn. If Algorithm 2C returns FALSE for any component, the vectors are judged to be not equivalent. Otherwise, if Algorithm 2C returns TRUE for each component, the vectors are judged to be equivalent.

In the second additional question mode, the user is asked to enter a set of vectors. The user response set is considered correct if it contains a vector equivalent to each vector in the correct answer set, and if the correct answer set contains a vector equivalent to each vector in the user response set. Since we already know how to compare two vectors, implementing this test is straightforward. There is a little challenge to be found in optimizing the checks to require a minimum number of vector comparisons; we will not discuss such details here.

## 5 Performance and Evaluation

### 5.1 Possibility of Evaluation

As has been mentioned previously, the gateway testing system has been in use in the University of Nebraska-Lincoln math department for a number of semesters. For the duration of its existence, the full details have been recorded of every student response to every question on every test that has been taken. This gives us a database of thousands of actual student responses<sup>11</sup> that can be used in order to determine how effective our algorithm is in practice. This is a wonderful resource, but it has one major drawback: with all of those thousands of responses, we do not know for certain which responses are correct and which are in error. While we know the original grade given to each response, we are not certain that grade is correct. From the rather low level of complaints about the gateway system's grading we can infer that most of the recorded grades are correct. Since it would be a monumental task to try to regrade those thousands of responses by hand, we need to look for other solutions.

The solution that we have employed is to compare different versions of the interval-based algorithms with each other and with the older floating point algorithms. Wherever there are discrepancies between the results that the algorithms produce, those discrepancies are examined by hand. Note that this process is not infallible. In particular, there are some sorts of errors for which the newer interval-based algorithms and the older floating point algorithms are likely to produce identical incorrect results. For example, the expressions  $\text{abs}(1000-x)$  and  $1000-x$  would most likely be judged equivalent by both algorithms. This happens because the "interesting" stuff in these functions happens outside of the relatively small range of values that our random sampling produces for  $x$ . Despite these limitations, the database of actual student responses provides us with a way to effectively check how our implementation fares in real-world scenarios.

For purposes of establishing as good a benchmark as possible, we created a variation of the interval-based algorithm described in this paper. That variation, which we will call the interval "*super-parser*," is identical to the ordinary implementation except that it requires a much higher number of hits (250) and allows a higher number of samples (5,000) than the ordinary implementation. This parser only judges expressions as correct after they have been found to agree on 250 points with the correct answer. All evidence indicates that such a comparison is highly accurate.

There are several distinct sets of student responses which were used in this evaluation. The first of these, which we will call the "Calc-106" data set, consists of the responses recorded by all of the calculus 106 students over the fall semester of 1998. This data set contains 8,434 well-formed responses to questions asking for an arbitrary expression. Those questions require the student to enter an expression equivalent to the correct answer. The second data set, which we will call the "Calc-107" data set, consists of responses recorded for the calculus 107 students over the fall semester of 1998. This data set contains 7,092 well-formed responses to questions asking for an arbitrary expression. Of these, 2,771 responses were to questions where the student was to enter an equivalent expression to the correct answer, and 4,321 responses were to questions where the student was to enter an expression which could differ by an additive constant from the correct answer.

---

<sup>11</sup>In the interests of privacy, all identifying information (names, i.d. numbers, etc.) was removed from the data before it was put in use for this project.

## 5.2 Correctness

We compared the results of the simple floating point algorithm to the interval super-parser on the Calc-106 data set. Of the 8,434 responses in the data set, there were only 45 discrepancies between the algorithms. Each of the discrepancies was a student response which had been graded as correct by the old floating point algorithm, but was graded as incorrect by the interval algorithm. When each of these discrepancies was hand graded, the result was that the interval algorithm was correct in each case. When the ordinary (requiring only 14 hits) version of the interval parser was run on the data set, it agreed with the interval super-parser on each response.

The first conclusion that we can draw from this examination is that the interval algorithm improves over the old algorithm. At every point where the interval algorithm and the simple floating point algorithm disagreed, the interval algorithm was correct. Another conclusion we can draw is that the simple floating point approach was performing quite effectively on this data set—the 45 misgraded responses account for only one half of one percent of all responses. Furthermore, all of those misgrades were in the direction of leniency, grading an incorrect response as correct. Of course, we do not know that we have found all of the misgraded responses in that data set. While we do have the theoretical guarantee from the design of the interval algorithm that there should be no correct responses which were graded as incorrect, in practice this guarantee relies on some pretty large assumptions. We are required to assume that Java’s math implementation is free of computational error, that we managed to write bug-free code, that the author of the `ia_math` package managed to write bug-free code, and that we did not make any subtle mistakes in the design of our algorithms. The real-world evaluation does a lot to suggest that none of these assumptions is seriously violated.

When we compared the results of the simple floating point algorithm to the interval super-parser on the Calc-107 data set, there were 158 discrepancies between the algorithms. Five of these were discrepancies where the simple floating point algorithm graded a response as incorrect that the interval algorithm graded as correct, and the remaining 153 were cases where the floating point algorithm graded a response as correct that the interval algorithm graded as incorrect. A hand grading of half of the discrepancies showed the interval algorithm to be correct in each case. In each of the five discrepancies in which the interval algorithm graded the response correct, the answer was indeed correct. These five responses, which were all on the same question, show that the old algorithm did not in practice maintain only one-sided error.

The vast majority of the discrepancies were the result of one difference between the interval algorithm to check equivalence up to an additive constant and the older floating point algorithm to do that. (We have not described the old floating point algorithm to check equivalence up to an additive constant in this thesis.) That difference is that the older algorithm counted as “wide” any trial where one of the expressions fails to exist, while the interval algorithm counts these trials as a “miss” when one expression is *certainly NaN* and the other certainly does exist. This strongly affects the grading of integration questions which produce the natural logarithm as part of their result. The correct indefinite integral of  $1/x$  is  $\ln(|x|) + C$ . A common student mistake<sup>12</sup> is to omit the absolute values and say the integral is  $\ln(x) + C$ . These formulas are not equivalent; the former exists as a real number for  $x < 0$  while the latter does not.

Comparison of the ordinary (requiring only 14 hits) interval parser to the interval super-parser on the Calc-107 data set produced one to two discrepancies on different runs<sup>13</sup>. The discrepancy that most consistently occurred (about half of the time) was a response to a question requiring equivalence up to an additive constant. The correct expression was  $2 \ln(x-1/3) - 2 \ln(x)$  while the user entered  $2 \ln(\text{abs}(3x-1)) - 2 \ln x$ . This differs from the correct expression in two ways. For most values of  $x$ , the only difference is that of an additive constant of  $2 \ln(3)$ . The absolute value function in the user expression, however, causes that expression to exist on the interval  $[0, 1/3]$  where the correct expression does not exist. The other discrepancy which occurred less often than one time in ten was also a response to a question requiring equivalence up to an additive constant. The correct expression was  $e^{(x^5)}/5$  while the user entered  $1/5(e^{(x^5)})x - 1/30(e^{(x^5)})x^6$ . The difficulty is that except for small values of  $x$ , both expressions produce such huge results that intervals a few ULPs wide are enormous. For  $x > 3.72$ , the expressions even exceed the value of the largest non-infinite floating point number. For  $x < -3.76$ , the expressions are smaller than the value of the smallest non-zero floating point number. On rare occasions, our sample points lie far enough out that we reach 14 trials before coming to the conclusion that the expressions cannot differ by a constant.

After careful consideration, we decided to allow these discrepancies to persist. It is the nature of the algorithm that

---

<sup>12</sup>The mistake is also common to calculus exam writers. Many of the “correct” answers in our question database were found to suffer from this same error.

<sup>13</sup>That the results vary from one run to another is not surprising — the algorithm is after all only probabilistic.

occasional incorrect responses will be graded as correct. To try to decrease the occurrence of this even farther could most easily be done by increasing the number of hits we require to decide equivalence. We decided against doing so because the gains in our algorithm's ability to discriminate between correct and incorrect responses seemed to be unduly small compared to the increased cost of run time to carry out more trials.

## 5.3 Speed

Through much of this thesis, the discussion has focused on the problem of correctly comparing expressions. We have paid only scant attention to one aspect of the problem which is of great importance to the usefulness of these algorithms in an online testing system. In order to be useful, an algorithm must be fast. The gateway system can in peak usage serve as many as 200 hits per minute as users' web browsers request documents and images. The system is dealing in real time with a lot of users. Fortunately, final submissions of student exams for grading is a very tiny part of the traffic that the system sees. Still, when a student exam comes in to be graded, the system cannot afford to spend a lot of time carrying out that grading.

A quick glance at the interval grading algorithm shows that the time it takes is directly proportional to the number of trials which are carried out before termination conditions are reached. This gives us a strong incentive to decrease the termination thresholds as low as is reasonable, and in particular to require only a small number of hits. It has been mentioned previously that the decision to require 14 hits was made based on empirical data. A test was conducted using the Calc-106 data set and a variation of the interval parser<sup>14</sup>. In this test, we counted the number of hits received on any expression which was eventually found to be incorrect. The most hits found on any incorrect response was 7; to introduce a margin of safety we doubled this value to 14. As we remarked above, this margin of safety was not large enough to avoid all errors in the Calc-107 data set, but it does keep the number of (detected) errors to a comfortable level.

The actual speed at which expressions are compared is quite acceptable. We have estimated that the system should not spend more than about half a second grading a ten-question calculus exam, and a quicker response than that would be even better. We ran time trials of the algorithm on a 200 MHz Pentium machine running the Windows 95 operating system and the JDK 1.1.7 Java virtual machine. For the Calc-106 data set, the average time spent to evaluate each expression was 0.031 seconds. Ninety-nine percent of the expressions were evaluated in under 0.27 seconds, and the maximum time spent to evaluate an expression was 0.83 seconds. In comparison, the old floating point routines averaged 0.039 seconds per expression. The speed improvement is largely a result of program optimization, not a difference between the underlying algorithms.

## 6 Conclusions

### 6.1 Future Work Possible

There are a number of ways in which the algorithms that we have developed could be improved. The current implementation chooses random variable assignments based on a Gaussian random distribution with mean 0.0 and standard deviation 10.0. Almost all of our trial points are chosen within a relatively narrow range of values. This means that our algorithms do poorly at discriminating between functions which agree on values within that range but disagree outside of that range. For example, we have already mentioned that  $\text{abs}(1000-x)$  would not be distinguished from  $1000-x$ . There is a slightly different problem for expressions that are defined only on a small subset of the range of values in which our trial points usually land. In order to find enough hits on an equivalent pair of these expressions, we must examine a very large number of trial points. In each of these cases, the algorithm would benefit from some method of better locating the area in which an expression has interesting behavior and concentrating trial points there. In the latter case, the interesting behavior of existence is easy enough to identify. For that case, a possible approach might be to have the mean and standard deviation of the Gaussian distribution change to reflect a running mean and standard deviation of the samples which were found where the correct function exists. This would produce a "crawling hump" as the distribution of trial points repositions itself over the range where the correct expression exists. A more

---

<sup>14</sup>The actual variation employed in the test used a different distribution for the random assignments than we have described. Namely, it assigned values to the variables based on a uniform distribution over the range from -25 to 25.

general solution to the problem would be to allow the question writer to specify interesting regions to sample, or just to provide “hints” as to where good sample points might lie.

Another potential area for improvement is the `ia_math` library that we make use of. There are certainly speed improvements possible within the library. Perhaps more significantly, the library could be improved to provide better results in the face of machine implementations of functions that suffer from computational error. As we mentioned in section 2.3, there is good reason to mistrust Java’s implementations of elementary functions. There are ways in which the library could be made more robust to such errors without losing efficiency.

## 6.2 Summary of Results

This thesis has presented work that has been done to develop, implement, and evaluate routines to probabilistically check the equivalence of expressions in an online testing environment. The algorithms that we have developed make use of a rounded interval mathematics scheme that is not new, but that has not (to the best of our knowledge) previously been applied to this problem. This work has resulted in significant improvement over the algorithms that were previously in use in the Gateway Web Testing System.

We have developed two closely related algorithms. The first of these provides a probabilistic solution to the problem of determining whether or not two expressions are equivalent. The second of these provides a probabilistic 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 which does not suffer from computational error in its elementary math routines), though in practice we choose to violate the one-sidedness in certain situations. 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.

We have implemented these algorithms within the Gateway Web Testing System and made a careful evaluation of their effectiveness when applied to actual expressions submitted by students on calculus exams. The new algorithms are a definite improvement over the algorithms previously in use. On every response where the new algorithms disagree with the old, the new algorithms have been found to be correct. Furthermore, the new implementation shows roughly a 27 percent speed improvement over the old implementation.

## References

- [1] Brewer, K. J.: *IEEE-754 References*; <http://babbage.cs.qc.edu/courses/cs341/IEEE-754references.html> (1999). Accessed March 8, 1999.
- [2] Gonnet, G. H.: Determining Equivalence of Expressions in Random Polynomial Time, *Proceedings of the 16th ACM Symposium on the Theory of Computing*, pp. 334-341 (April 1984).
- [3] Gonnet, G. H.: New Results for Random Determination of Equivalence of Expressions, *Proceedings of the 1986 ACM Symposium on Symbolic and Algebraic Computing*, pp. 127-131 (July 1986).
- [4] Hammer, R., Hocks, M., Kulisch, U., Ratz, D.: *C++ Toolbox for Verified Computing*; Springer-Verlag, Berlin (1995).
- [5] Hickey, T. J.: *ia\_math library version 0.1beta1*; [http://www.cs.brandeis.edu/~tim/Packages/ia/ia\\_math/](http://www.cs.brandeis.edu/~tim/Packages/ia/ia_math/) (1997). Accessed December 24, 1998.
- [6] Institute of Electrical and Electronics Engineers: *IEEE Standard for Binary Floating-Point Arithmetic*; IEEE (1985).
- [7] Intel Corporation: *Intel Architecture Software Developer’s Manual Volume 1: Basic Architecture*; <http://developer.intel.com/design/pentium/manuals/24319001.pdf> (1996). Accessed March 8, 1999.
- [8] Juffa, N.: *Everything You Always Wanted to Know About Math Coprocessors, Version 6*; <ftp://x2ftp oulu.fi/pub/msdos/programming/docs/copro16a.zip> (1994).

- [9] Martin, W. A.: Determining the Equivalence of Algebraic Expressions by Hash Coding, *Journal of the ACM* **18**(4) pp. 549-558. (Oct., 1971).
- [10] Moore, R. E.: *Interval Analysis*; Prentice-Hall (1966).
- [11] Orr, J. L.: “Wiley Web Tests for Calculus”; John Wiley & Sons, New York. (1998).
- [12] Orr, J. L.: “Wiley Web Tests for Precalculus & College Algebra”; John Wiley & Sons, New York. (1999).
- [13] Richardson, D.: “Some Undecidable Problems Involving Elementary Functions of a Real Variable”; *Journal of Symbolic Logic*, Vol. 33, No. 4 (Dec., 1968), pp. 514-520.
- [14] Sun Microsystems: *Java Platform 1.1.5 Core API Specification: Class java.lang.Math*; (1996).
- [15] Sunsoft: *Freely Distributable LIBM, Version 5.2*; <http://www.netlib.org/fdlibm/> Accessed January 9, 1999.