Harmonic Sum Calculation: Sneaking Finite Precision Principles into CS1

Andrew A. Anda
Department of Computer Science
St. Cloud State University
aanda@eeyore.stcloudstate.edu

Abstract

We describe using the calculation of harmonic sums to introduce and integrate the discussion and exploration of some elementary principles of finite precision floating point computation into a CS1 course. After students' exposure to this problem and its solutions, they should not only have a better understanding of when to exploit event-controlled vs. counter-controlled loops, but also be able to correctly answer the following two questions relating to operations on floating point data:
1. can $(A + B) + C \neq A + (B + C)$ for some floating point A, B, C?
2. can $(A + B) = A$, where $|B| > 0$ for some floating point A, B?

Introduction

Numerical computation consists essentially of the study, and implementation in software on some architecture, of those algorithms which compute via finite precision (usually floating point) arithmetic. However, there has been a progressive process of reduction in numerical computation content in the sequence of ACM Computing Curricula recommendations. In fact, the current 2001 Report (The Joint Task Force on Computing Curricula – IEEE Computer Society, ACM, 1991) has eliminated from the core curriculum all seven lecture hours of *numerical and symbolic computation* recommended by the 1991 Report (Tucker, A.B., et al, 1991). An understanding of at least a few basic principles of numerical computation is essential for anyone who performs arithmetic operations on floating point data.

Floating point computation is pervasive and unavoidable in such diverse disciplines, and applications to disciplines, as economics and finance, accounting, applied mathematics, applied statistics, and just about all of the engineering and science (hard and soft) disciplines. While unnecessary degrees of inaccuracy and inefficiency are common (e.g. statistics, computational geometry, and modeling of continuous

systems in general), there are several well known anecdotes and examples of catastrophic failure attributed to floating point computation related factors: e.g. the Patriot missile failures, and the Arian rocket failure. (Over:01, 2001).

The undergraduate computer science major or minor should certainly expect to have some exposure to the theory and practice of computing with floating point data, but often this takes place only in hardware or architecture related coursework where the connection to the behavior of implemented algorithms may not be emphasized. However, many of the students in a CS1 course are majors in other disciplines, the majority of which are quantitative. For these students, CS1 may their only formal exposure to computing theory and practice, yet they will be expected to perform floating point computation as part of their further studies in their discipline and/or the practice of their profession.

The CS1 curriculum is usually fairly well defined at each educational institution where it is taught. The current ACM Computing Curricula report (The Joint Task Force on Computing Curricula – IEEE Computer Society, ACM, 1991) specifies several of the more popular implementation strategies: imperative, objects, functional, breadth, algorithms, and hardware first. Most include programming, and more specifically, loop control constructs. It is those CS1 courses that I will be addressing. Because there is often little latitude for modifying the curricular content of these well defined courses (as well, if one adds some new content to the course it is usually necessary to determine and excise some other content), The only practical option is to *sneak in* new content by overloading some aspect of the existing content. This paper describes the overloading numerical content onto the practice of loop control.

Problem for Student to Solve

A harmonic series is the sum of a sequence of terms, where the terms are the reciprocals of all of the set of positive integers:

$$H_n = 1 + 1/2 + 1/3 + \ldots + 1/n = \sum_{k=1}^{n} 1/k, n \geq 0. \quad (1)$$

H_n for any given finite n is also labeled a *Harmonic number*.

In mathematical theory, using *infinite precision* arithmetic:

1. the infinite sum of a harmonic series is infinite;
2. it grows as slowly as a log function;
3. the *order* in which the terms are summed is irrelevant (the *associative* law).

However, in computing practice, using finite precision arithmetic, the *order* in which the terms are summed affects the *accuracy* as well as, for some orderings, the *number of terms* which may be effectively summed. The greater the difference in magnitude between the two operands of a floating point sum, the more *information*, or *significance*, is lost from the smaller when added to the larger in the process of

aligning the decimal point. This is one example of a *roundoff error*. Most of the significance of a small number is lost when it is added to a large number. When the *ratio* of the larger number to the smaller number becomes *sufficiently* large, the value of the sum will be *identical* to the value of the larger number. However, we can reduce or limit this kind of roundoff error by adding terms of close relative magnitude where the magnitude of the ratio of the two operands is close to unity, thus serving to reduce, if not minimize, the number of places that the decimal point must be shifted.

Even without rearranging terms, the number of possible ways to sequence the respective additions grows exponentially with the number of terms (more specifically, proportional to the Catalan numbers, $C_n = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix}$). There are a significant number of established ordering algorithms which have been analyzed both in the context of accuracy and complexity. (See (Hig:02, 2002) for the most recent survey and relevant bibliography.) We will focus only on the two *recursive* algorithms:

1. the *forward* sum (summing from largest to smallest)
2. the *backward* sum (summing from smallest to largest)

(**Note:** We don't have to use explicit functional recursion, iteration can be used as well)

As the forward sum progresses, the partial sum increases as each subsequent term to be included in the sum decreases. This is not desirable. It will cause a progressive worsening of the cumulative effects of the roundoff errors as they become more extreme.

The other problem that will occur is that, starting with some term, that term and all subsequent terms will cease to contribute to the sum. This can be explained by considering a decimal point shift of the smaller term so extreme that there is no overlap between the significant digits of the larger term and the significant digits of the smaller term, thus the smaller term cannot contribute to the sum. The backward sum, however, maintains a much better ratio between the operands of each binary sum because the magnitude of the successive terms are growing along with the running partial sum.

There are three relevant bit fields of a radix 2 floating point number:

sign (one bit),

significand (usually *normalized* with an implicit leading bit),

exponent (*biased*).

The number of bits in the significand determine the precision of the number, i.e. for example how many values may be exactly represented in the interval $[0, 1)$. The number of bits in the exponent field limits the range of values, i.e. how large or small in magnitude can a representable number be.

There are a variety of important constants, derivable from the number of digits of the significand and range. One of the most important is *machine epsilon* which

represents the difference between 1.0 and the next larger representable floating point number. (Over:01, 2001) Another way to characterize *machine epsilon* is that it is the smallest floating point number A such that $A + 1.0 \neq 1.0$.

When the ratio of a new term to the partial sum becomes smaller than the *machine epsilon* for the working precision, the term will not contribute to the sum. The *machine epsilon* value is a function of the number of digits of the significand. For the forward sum, we will attain a small enough ratio for some iterate. It is at that point that we will terminate the forward loop. We can then use the stopping term of the forward sum as the starting term of the backward sum. We can then compare the results.

Our computer language will usually provide a set of intrinsic fundamental floating point types, If there are two or more, we can compare the forward and backward sums in each precision to the backward sum in the highest precision. Additionally, we can compute a closed form expression to approximate H_n: (Knuth, 1973)

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \epsilon, 0 < \epsilon < \frac{1}{252n^6}. \tag{2}$$

Where γ represents *Euler's constant* and has the value,

$$\gamma = 0.44742147706766606172232157437601002513132552\ldots$$

(**Note:** The first forward sum which determines the number of terms to sum for all others must be performed in the type with the smallest precision, or it may not terminate in a reasonable amount of time. Additionally, an integer loop counter type of sufficient range must be selected.)

Objectives

This content may be presented in a lecture, lab, homework assignment, module, or as some synergistic combination thereof.

I have found it helpful prior to working through this exercise to have the students work through a lab exercise wherein they print out the set of constants which characterize all of the integer types and floating point types for computing environment (language, architecture, and compiler) that they are using. In C++, these are the constants from `climits.h` and `cfloat.h`. These constants include specifications of significand size, precision, and range which will be relevant in the current exercise. This prior lab also provides additional exposure to the topics of differing sizes for the same basic data type.

Ostensible Objectives

The *ostensible* objective of this exercise is to provide the student with an example of when they should use an *event-controlled* loop vs. a *counter-controlled* loop. Because the student doesn't know in advance the point at which the terms in the

forward sum will cease contributing to the partial sum, they need to use an *event-controlled* loop. When the sum stops changing, we exit the loop, and subtract one [1] from the running count of terms we have added. Once the student has determined the number of terms contributing to the sum, they can then use *counter-controlled* loops for the other sums. The challenge for the student will be the crafting of the event-controlled loop. The first decision the student must make is to decide between the pre-test and post-test variants.

One example of this event-controlled loop in the **C++** language is:

```
do
    {
    oldsum = newsum;
    newsum = oldsum + 1.0f / ++terms;
} while (oldsum != newsum);
ffsum = newsum;
```

In my experience, students encounter difficulties getting the logic of the above loop correct. I find it helps to hint about using an update expression such as
newValue = oldValue + smallCorrection.
Even with that hint, the average student will wrestle some with the logic.

Numerical Objectives

The principal numerical objectives of this exercise is to prepare the student to answer correctly the following two questions relating to operations on floating point data:

1. can $(A + B) + C \neq A + (B + C)$ for some floating point A, B, C?
2. can $(A + B) = A$, where $|B| > 0$ for some floating point A, B?

Beyond answering these two questions, students receive exposure and practice with the numerical effects of changing the floating point range and precision.

If one explains the representation of the floating point number, the student can learn about two new ways to to complement integers

bias used for exponent – only one zero,

sign magnitude used for significand – two zeros,

in addition to those for binary integers they are usually exposed to

one's complement not too common – two zeros,

two's complement common – only one zero.

The IEEE 754 Floating Point Standard (IEEE, 1987; Kahan, 1995; Schwarz, 2003) should be at least mentioned as it is now implemented on virtually all modern systems. Much of the following can bleed into an architecture and systems course

where this exercise may be extended and generalized to help demonstrate the relationships between hardware, systems, and languages. The following significant topics may be discussed in the context of the IEEE 754 standard and its implementation:

rounding rounding to infinite precision using two guard bits and a *sticky* bit;

rounding modes rounding to $\pm\infty$, 0, and nearest;

exact results not all floating point operations are inexact. A significant number are exact for a certain set of operands;

± 0 benefits and concerns regarding having two zeros;

5 exception flags invalid, division by zero, overflow, underflow, and inexact;

exception values NaN, $\pm$Inf, subnormals – operations on and representations;

extended formats Implemented on some architectures such as the Intel X86 line. I only know of double extended to be implemented. The standard leaves a lot of latitude for differences among the implementations. (Priest, 1998)

FMAs & short vector SIMD functional units (Nievergelt, 2003)

Exploiting IEEE 754 exceptions A new software engineering paradigm for numerical algorithms is to run the more robust (usually rescaling) method only after catching a numeric exception; (Demmel & Li, 1994)

Benchmarking can provide additional information as well. If CPU timers bracket each loop, the students can learn about the temporal effects of

loop overhead

different operations (add a loop variant which substitutes a multiplication for the division – this will exhibit a significant difference on some architectures),

different precisions of the same type. Some architectures and compilers require much more time for the longer precisions, e.g. long double in C++ on some Sparc architectures.

function call overhead (add a loop variant which substitutes a function call to the operation, and the restorative effect of inlining)

different languages Some languages impose a procrustean transformation on the appearance to the user of the underlying architecture. E.g. Java requires all underlying architectures to function like a Sparc (i.e. 64 bit double precision only and no use of fused multiply-add units [FMAs]), resulting in significantly reduced speed and accuracy on some architectures such as Intel Pentium. (Kahan, 1998)

different data On certain platforms, differing time penalties are imposed on operations involving certain IEEE 754 exceptional values, e.g. NaN, Inf, subnormal. (Beebe, 2000)

optimization level The various optimization levels can have varying (usually positive) effects on the timing and the accuracy as well (e.g., allowing for the use of FMAs or short vector SIMD instructions). (Nievergelt, 2003)

Closing

Certainly there's enough important information regarding floating point, and more generally, numerical computation to fill at least an entire undergraduate course devoted to this topic alone. (Anda, 2003) However, in the context of this project, I would hope that the at the conclusion of their CS1 course, students would take with them at least an understanding that when they compute with floating point data they should be alert to the possibility of *weird* behavior. Being forewarned, they could then either proactively or reactively acquire the additional knowledge to better understand the effects of their floating point computations.

I have outlined some extensions to the basic format of this project. I encourage the use of these and other extensions of this project in subsequent computer science or computational science courses beyond CS1 to assist students towards a better understanding of floating point computation.

References

Anda, A. A. (2003). High-performance numerical computation: A proposal of a quasi-capstone course for csci majors. In T. Gibbons (Ed.), *Midwest instructional and computing symposium.* CS/CIS Dept., College of St. Scholastica.

Beebe, N. H. F. (2000). *The cost of IEEE 754 exceptional operands and instructions* (World-Wide Web document). Salt Lake City, UT 84112, USA: Center for Scientific Computing, Department of Mathematics, University of Utah.

Demmel, J. W., & Li, X. (1994). Faster numerical algorithms via exception handling. *IEEE Transactions on Computers*, *43*(8), 983–992.

Higham, N. J. (2002). *Accuracy and stability of numerical algorithms* (2nd ed.). SIAM.

IEEE. (1987). IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, *22*(2), 9–25.

Kahan, W. (1995, May). *IEEE standard 754 for binary floating-point arithmetic.* (Lecture Notes in progress)

Kahan, W. (1998). How Java's floating-point hurts everyone everywhere. In ACM (Ed.), *ACM 1998 workshop on java for high-performance network computing* (pp. ??–??). New York, NY 10036, USA: ACM Press.

Knuth, D. E. (1973). *The art of computer programming, vol. 1* (2nd ed.). Reading, MA: Addison-Wesley.

Nievergelt, Y. (2003). Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Transactions on Mathematical Software*, *29*(1), 27–48.

Overton, M. L. (2001). *Numerical computing with IEEE floating-point arithmetic.* SIAM.

Priest, D. (1998, January). *Differences among IEEE 754 implementations.*

Schwarz, E. (2003). Revisions to the IEEE 754 standard for floating-point arithmetic. In IEEE (Ed.), *16th IEEE Symposium on Computer Arithmetic: ARITH-16 2003: proceedings: Santiago de Compostela, Spain, June 15–18, 2003* (pp. 112–112). 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA: IEEE Computer Society Press.

Sun Microsystems. (1991). *Numerical computations guide.* (Sun Microsystems, 1991. Numerical Computations Guide, Document number 800-527710.)

The Joint Task Force on Computing Curricula – IEEE Computer Society, ACM. (1991). *Computing curricula 2001; computer science; final report.* Los Alamitos, CA: IEEE Computer Society.

Tucker, A.B., et al. (1991). *Computing curricula 1991: Report of the acm/ieee-cs joint curriculum task force.* New York: Association for Computing Machinery.