WHEN e IS REALLY π

E. Gomez[1], Y. Karant[1], K.E. Schubert[1]
{egomez, ykarant, schubert}@csci.csusb.edu
Department of Computer Science
California State University
San Bernardino, CA 92407

ABSTRACT: *Floating point numbers are the standard way of approximating the real numbers, but they lack many of the nice properties such as associativity, commutativity, and completeness. Many ramifications of this are well documented in elementary numerical analysis texts, such as arbitrary errors in calculations. In this paper, a different error will be examined, that being when calculations approximate a calculation technique for a known constant, and thus generate a near exact computation of the wrong constant. In particular, a technique will be shown that will calculate e to arbitrarily many digits, when in the real numbers it should result in π. The technique is simple to understand and provides an excellent pedagogical tool to introduce the concepts and dangers of floating point numbers and their attendant computations.*

KEYWORDS: *Numerical Analysis, Education*

1. INTRODUCTION

Due to the speed and ease of use of floating point calculations, they are the primary calculation technique used in scientific calculations. Usually floating point calculations produce excellent results, but in some cases the inherent differences between floating point numbers and the real numbers generate unrecoverable errors. A favorite technique of most numerical analysts is to show that arbitrary differences can result when certain conditions apply. In particular we will consider when the result of a calculation should be π, but it ends up an equally famous constant, e. To distinguish a calculation done in the real numbers from one done in floating point, the notation $fl(\bullet)$ will be placed around any floating point calculation in a math equation.

The outline of the paper is as follows. First, the basic problem will be introduced in Section 2. Second, a mathematical construction proof will be outlined showing that the result must happen regardless of the precision in Section 3. Third, the difference between the function pow and repeated multiplication will be examined in the scope of this problem in Section 4. Then the pedagogy of the problem will be explored in Section 5.

2. PROBLEM STATEMENT

Consider the following problem on the real numbers.

$$y = \sqrt[2^{52}]{\pi} \quad (1)$$

$$z = y^{2^{52}} \quad (2)$$

Since this is on the real numbers it is trivial to see that $z = \pi$. Computers do not calculate in real numbers though, they usually approximate the reals by floating point numbers, such as IEEE 754 standard, which will be used here. The problem can be coded in C as Fig 1.

When compiled and run it gives the answer

```
The answer is 2.7182818284590451
e is          2.7182818284590451
The answer is e!
Press any key to continue
```

The answer ends up as e to the full precision of the number, every bit is the same! Should such a result be generated it would not be doubted, so what happened and how can we handle it.

[1]The authors gratefully acknowledge the support of the NSF under award CISE 98-10708

```
#include<cmath>
#include<iostream>
#include<iomanip>
using namespace std;

int main(){
    double pi,y,z,e, tol=.0000000000000001; // 10^{-16}
    pi=4*atan(1); // pi to full precision
    e=exp(1);     // e to full precistion

    y=pow(pi,pow(2,-52)); // Eq. 1
    z=pow(y,pow(2,52));   // Eq. 2

    cout << setiosflags(ios::showpoint | ios::fixed) << setprecision(16);
    cout << "The answer is " << z << "\n";
    cout << "e is          " << e << "\n";
    if(abs(z-e)<tol)
        cout << "The answer is e!"<< "\n";
    return 0;}
```

Figure 1: C code for Pi2E

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main(){
    double pi, e, result;
    int i;
    e=exp(1);
    pi=atan(1)*4;
    result=pi;

    for(i=1;i<53;i++)
        result=sqrt(result); //Eq.1
    for(i=1;i<53;i++)
        result=result*result; //Eq.2

    cout << setiosflags(ios::showpoint | ios::fixed) << setprecision(16);
    cout << "The answer is " << result << "\n";
    cout << "e is          " << e << "\n";
    return 0;}
```

Figure 2: C code with for loops instead of pow

3. GENERATING e

There are many ways to calculate e. The one of relevance to us is to calculate

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n. \quad (3)$$

On a floating point machine, the best that can be done due to precision constraints is to use $1+\varepsilon$ for $1+\frac{1}{n}$ as by definition of ε, $1+\varepsilon$ is the closest number to 1 that is greater than 1. The value of ε can be calculated for any implementation by using the definition, by continually adding smaller and smaller values to 1 and taking the last one that generates a result that isn't 1. This can be simplified to looking at the representation of the floating point number and setting the bit in the least significant location of the significant (note the significant is also called the mantissa in math nomenclature). Using this later technique implies in IEEE double precision that $\varepsilon = 2^{-52} \approx 10^{-16}$, and thus

$$e \approx fl\left(fl(1+2^{-52})^{2^{52}}\right). \quad (4)$$

Now the only part that remains is to generate $1+\varepsilon$ from π. To do this, note that square root is a contraction with fixed point at 1. Thus in the real numbers, for any number, $x > 1$, there exists a δ such that $x > 1+\delta > \sqrt{x} > 1$. By repeatedly applying the square root contraction the result can be brought as close to 1 as needed. For π, the initial value of $\delta < .78125_{\text{base }10} = .11001_{\text{base }2}$, and each successive square root approximately halves δ, essentially shifting the binary radix point by 1. For double precision the radix point needs to be shifted 51 places to achieve $\delta = 2^{-52}$, so a total of 52 square roots must be done or equivalently calculating $\pi^{2^{-52}}$. Combining everything yields that on a double precision floating point computer

$$e \approx fl\left(fl\left(\pi^{2^{-52}}\right)^{2^{52}}\right). \quad (5)$$

It is trivial to extend this to other representations. The general formula is

$$e \approx fl\left(fl\left(\pi^{2^{-n}}\right)^{2^n}\right), \quad (6)$$

where n is the number of bits in the significant. Note that Eq 5 and Eq 6 in the real numbers would give the result $e \approx \pi$, which is a bad approximation. In double precision floating point numbers, however, the approximation is good to all 53 bits of the significant (52 bits in the representation plus the hidden bit for normalized results) for double precision, or $n+1$ in general if hidden bits are supported. If the hidden bit is not supported then replace n by $n-1$ everywhere (including the algorithm).

In particular, consider IEEE 754 single precision floating point numbers, which have 23 bits in the significant and a hidden bit. The algorithm is thus

$$y = fl(\sqrt[2^{23}]{\pi}) \quad (7)$$

$$z = fl(y^{2^{23}}) \quad (8)$$

When coded in C and run it yields

```
The answer is 2.7182817
e is          2.7182817
The answer is e!
Press any key to continue
```

Again the algorithm performs as expected.

4. REPEATED CALCULATIONS

So far the function pow has been used to calculate x^{2^n}. IEEE 754 requires that every operation on the floating point numbers must be accurate to the least significant bit of the representation. Since pow was written to this specification it yields an answer between $(1+\varepsilon)e$ and $(1-\varepsilon)e$. It is worth noting that x^{2^n} can also be calculated by a for loop:

```
y=x;
for(i=0;i<n;i++)
    y*=y;
```

If the for loop version is used each individual operation has the calculation bound, which means in essence that up to 1 bit of precision can be lost with each calculation. This method is encoded in Fig 2. When run it generates the output

```
The answer is 2.7182818081824731
e is          2.7182818284590451
Press any key to continue
```

Notice that it generates a less precise calculation, both from the floating point result of approximately e (it is accurate to only 8 decimal places instead of 16) and from the real number result of π ($|\pi - e| < |\pi - 2.7182818081824731|$).

5. PEDAGOGY

This problem has been used at CSUSB to introduce floating point calculations, and the attendant errors. Several benefits of using this problem as a case study for students learning floating point numbers are outlined below, they form the basis of a good use of the problem in the classroom.

1. Students can readily see the importance of distinguishing e and π in calculations.

2. The method of calculating e used in Eq 3 nicely ties together the basic calculus course with the course in numerical analysis or machine organization it is being taught in. Many Computer Science students do not understand how the math courses they take relates to the work they do. This gives them one example where it is easy to see the connection.

3. One of the major areas of work in numerical analysis is in the calculation of e^x. This problem introduces a method of calculating e and some of the issues around maintaining precision in the calculation of constants and functions by different techniques.

4. The use of square root as a contraction mapping can show students how truncation (by rounding or chopping) works in floating point numbers. It easy easy to see that something that contracts to a point will be sensitive to truncation as the bits that will carry the distinction between two different results of the contraction will be in less significant positions.

5. Dangerous error buildup is illustrated by the difference between using pow and a for loop. In most programmers understanding, there is no difference between pow and repeated multiplication in a for loop, but a clear difference is shown in the loss of 8 decimal places of accuracy.

6. As a side bonus the introduction of advanced math concepts (contraction mappings and fixed points) lead to many other easy to program problems of practical importance, like zero finding and optimization.

7. As a final area of benefit the students can see that this cannot be cleared up by adding more precision to the number. It brings into focus the inherent dangers of using floating point numbers and the need to know roughly what the answer should be before calculating. This final point cannot be overestimated as most students today have an inherent trust in the results of their programs, and rarely precalculate estimates for sanity checks.

The points in the above list are designed as a basis for issues to be brought up with students, and provide excellent opportunities for rigorous numerical proofs, practical programming discussions, and even mini projects to enhance learning. The flexibility of recoding the computation areas should be utilized to get students looking not only at different ways to code, but to examine the properties of the techniques and how to prove them.

The inherent dangers of floating point calculations can provide a wonderful chance to discus other techniques of handling scientific calculations, that rarely are discussed, such as interval arithmetic. The trade offs associated with the representation and computation of results can then be addressed in both theoretical and practical design techniques.

6. SUMMARY

Even among people familiar with numerical analysis and the arbitrary errors possible in floating point calculations, few would anticipate that a result of a simple calculation would be π in the real numbers and e if calculated in floating point numbers. This example forms a great introduction to the concepts and perils of calculating on a computer. The large number of issues that the problem touches allows it to be used as a unified connector of concepts and course material. The problem is also very memorable so students tend to remember it, and the concepts it introduces long after many other results are forgotten, and has even inspired a few students to continue studies in numerical techniques.