Ph.D. Coursework Tutorial Report

Error in Floating Point Computations

Ishwar Singh

Submitted to: **Prof. Awadesh Prasad**

November 10, 2020

Department of Physics and Astrophysics
University of Delhi
New Delhi, India

Contents

List of Figures

1 Introduction

Recently, a new category, computational physics, has been added to the traditional physics classification scheme. This new category acts as a bridge between the traditional theoretical and experimental physics. Computer simulations have become an integral part of not only every branch of physics but also in other disciplines of science. With each advancement of computer technology, the usage of computers for scientific computation has also increased substantially.

Although, computers are very powerful, but they have limitations. The representation of real numbers in computers is one of the most import limitations of computers. The computer is a finite system. This inability of the computers i.e. to store 'exact' real numbers, gives rise to the errors in floating point computations.

These errors, although small ($\epsilon \approx 10^{-7}$ for single precision and $\epsilon \approx 10^{-15}$ for double precision), propagates after every usage, might lead to unexpected results (known as garbage values). Therefore, it is imperative to study the errors involved in scientific computations before actually jumping into the field which requires computations.

2 Representation of Numbers in Computers

Computers are binary machines i.e. they understands sequences of binary integers (known as bits) i.e. zeros (0's) and ones (1's). All computations are also done using these arrays of zeros (0's) and ones (1's). These long arrays of binary digits are good only for computers, but it is not user friendly. A user enters his data in decimal numbers and expects his answer in decimal numbers only. A group of these bits is collectively known as a word with a well defined length. The computers are very fluent in this (binary numbers) mode of communication. Different computers might have different word lengths, but this length is generally expressed in bytes, with

$$1 \text{ byte} \equiv 1\text{B} = 8 \text{ bits}.$$

This seems a right time to introduce the single precision and double precision numbers. A single precision number is stored in an array of 32-bits i.e. 4 bytes, while the double precision number is stored in an array of 64-bits are 8 bytes. These two terms will reoccur multiple times in this report.

2.1 Integers

It is easier for computers to deal with the integers. In total 32 bits are allotted to store the integers, out of which 31 bits stores a straight binary number and 1 bit is for sign. The largest number which can be store is,

$$(01111.....1111)_2 = 2^{31} - 1 = 2,147,483,647.$$

2.2 Real Numbers

Storing real numbers is a bit complex. There exists two possible ways of storing real numbers in computers : (i) *fixed point representation* and (ii) *floating point representation.* In the fixed point representation methods, a fix number of bits are reserved before and after the decimal point. This method was good for example banking system where 2 or 3 numbers are required after the decimal point, but wasn't good for scientific purposes. The floating point method is used for scientific computations.

In general, a fractional number, of precision p, base β and exponent e, is represented as,

$$\pm d_0.d_1d_2d_3.....d_{p-1} \times \beta^e,$$

which has the value,

$$\pm(d_0 + d_1\beta^{-1} + d_2\beta^{-2} + ... + d_{p-1}\beta^{-(p-1)})\beta^e.$$

The numbers before the base β, i.e. $\pm d_0.d_1d_2d_3.....d_{p-1}$ is known as the mantissa. For computers the base is fixed i.e. $\beta = 2$. The mantissa and exponents, depending upon the precision requires for a computation, are allotted different finite numbers of bits. In a single precision number (32-bits) the mantissa carries 23 bits, the exponent carries 8 bits and 1 bit is reserved for sign of the number. In a double precision number (64-bits), the mantissa carries 53 bits, the exponent carries 11 bits. Please note that in both cases the numbers of bits assigned to the manstissa and the exponent is finite. Generally, a bias is added in the exponent. The single precision floats have a bias of + 127. Therefore, the stored exponent is = true exponent + 127.

The IEEE standard also reserves some special arrangements of mantissa and exponents for special values like $\pm\infty$ and for NaN (not a number). Exponent 255 is reserved for these spe-

cial values. Table 1 shows few relevant specifications of single-precision and double-precision data types.

	float	**double**
bits	32	64
bytes	4	8
range	$\approx 10^{-38}$ to 10^{38}	$\approx 10^{-38}$ to 10^{38}
accuracy	7 decimals	16 decimals

Table 1: Specifications of single-precision (float) and double-precision (double) data types.

3 Errors in Floating Point Computations

Every measurement is prone to errors. Sometimes these errors are caused by the researcher's negligence, some measurement are prone to random errors. Similarly, some errors are introduced by the computers used to perform a scientific computations. This section will deal with different types of errors and uncertainties introduced due to computer computations.

3.1 Types of Errors

Let's assume that a computation task is finished after n complex loops or steps. Let p be the probability of success of each step, then the joint probability of the complete program/code is $P = p^n$. If $p = 0.9993$ i.e. very likely to be true, after $n = 1000$ steps, the probability of the code to return correct result is $P \approx \frac{1}{2}$ i.e. the result is equally likely to give garbage values (wrong results). The most dominant errors in floating point computations are discussed in the upcoming sections.

3.1.1 Bad Theory or Blunders

These errors are introduced due to typographical errors which might cause syntax errors in the codes, running a different program or having a fault in the reasoning behind the code, using data which has errors in itself which is not known to the programmer. These errors can only be resolved by the programmer itself. One of the best ways to avoid such errors is to verify if the code is doing what the programmer every now and then.

3.1.2 Random Errors

These errors are caused by random events such as fluctuations in electronic components of the system, cosmic ray interactions inside the system (very very unlikely though) or if someone pulls the plug of your system and the system shuts down. All these errors are very

unlikely for small codes or programs. But these errors might be relevant for a week long simulations. This error is irrelevant for this project.

3.2 Approximation Errors

Consider we will have to use $\sin(x)$ in our program. Most of programming language provides an in-built function which returns the values of these trigonometric functions. Mathematically these functions are actually an infinite series e.g.

$$\sin(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} (\text{ exact }).$$

But it is not possible to compute all the terms in these functions. Most of the algorithms compute these sums to a finite number of terms i.e.

$$\sin(x) \approx \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} (\text{ algorithm }), \tag{3.2.1}$$

$$= sin(x) + \epsilon(x, N), \tag{3.2.2}$$

where $\epsilon(x, N)$ is known as the *approximation error* which is given as,

$$\epsilon(x, N) = \sum_{N+1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!}.$$

As these errors arise from the algorithms used to approximate the mathematics, these errors are also known as *algorithmic errors*.