Ph.D. Coursework Tutorial Report

Errors and Uncertainties in Computations

Ishwar Singh

Submitted to: **Prof. Awadesh Prasad**

November 12, 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 special 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 Types of Errors

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. 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 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.

3.2 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.

3.3 Approximation Errors

Consider we will have to use $\sin(x)$ in our program. Most of programming languages provides an in-built function which returns the values of such 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.

$$\sin(x) \approx \sum_{n=1}^{N} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} (\text{ algorithm }), \tag{3.3.1}$$

$$= sin(x) + \epsilon(x, N), \tag{3.3.2}$$

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*. It seems reasonable to say that approximation errors decreases with the increasing value of N and vanish in the limit where $N \to \infty$. In general, the approximation errors, ϵ_{approx}, varies as,

$$\epsilon_{approx} = \frac{\alpha}{N^{\beta}}.$$

where α and β are arbitrary constants.

3.4 Round-off Errors

A finite number of bits is available with the computer (23 bits for single precision and 53 bits for double precision data types) to store the mantissa of any number stored in it. The float data type can hold numbers up to 7 significant figure while the double data type can hold up to 16 significant numbers. If a float type variable is asked to store a number with 10 significant digits, computer will store a rounded-off 7 significant digit number in it's memory.

The same happen with the double data type [1]. This rounding off the numbers introduces errors in the computations which grows up with each arithmetic operation. The computer stores an exact number 'a' is stored as 'a_c' such that

$$a_c = a(1+\epsilon_a),$$

where ϵ_a is the relative round off error in a.

3.4.1 Round-off Error in a Single Step

Division of two numbers, as calculated by the computer is given as,

$$\begin{aligned} a = \frac{b}{c} \Rightarrow a_c = \frac{b_c}{c_c} &= \frac{b(1+\epsilon_b)}{c(1+\epsilon_c)} && (3.4.1)\\ \Rightarrow \frac{a_c}{a} &= \frac{1+\epsilon_b}{1+\epsilon_c} \approx (1+\epsilon_b)(1-\epsilon_c) \approx 1+\epsilon_b-\epsilon_c && (3.4.2)\\ \Rightarrow \frac{a_c}{a} &\approx 1+|\epsilon_b|+|\epsilon_c| && (3.4.3) \end{aligned}$$

Notice however that in the above derivation very small ϵ^2 term has been ignored and the modulus is used to find the maximum error in the calculation. Note also that this is the basic error propagation rule which we use in our laboratory calculations.

3.4.2 Round-off Error Propagation After N Steps

As we have seen in the previous section, the round-off error accumulates after each calculational step. In general, If there are N calculational steps involved in an algorithm, the total relative error ϵ_{ro} becomes,

$$\epsilon_{ro} \simeq \sqrt{N}\epsilon_m.$$

where ϵ_m is the machine epsilon. Machine epsilon is the minimum number such that a system can differentiate 1.0 from $1.0+\epsilon_m$. In other words,

$$1.0+\epsilon_m > 1.0.$$

This is equivalent to the least count of any apparatus we use in our laboratory. This $\epsilon_m \simeq 10^{-7}$ and $\epsilon_m \simeq 10^{-16}$ for float and double data type respectively. Machine epsilon plays

[1] If the significant digits of number is greater than 16.

an important role in defining errors in scientific computations. This machine epsilon is the upper bound of the round off errors.

The fact that round-off errors accumulates with each computational steps, makes it necessary to study the variation of these errors with N. This helps the programmer to figure out when to stop the iteration.

3.4.3 Subtractive Cancellation

This is actually an arithmetic error which arises if two large and approximately equal numbers are subtracted, the relative error in the answer increases after each step. Consider the subtraction of two numbers,

$$a = b - c \Rightarrow a_c \simeq b_c - c_c \simeq b(1+\epsilon_b) - c(1+\epsilon_c) \tag{3.4.4}$$

$$\Rightarrow \frac{a_c}{a} \simeq 1 + \epsilon_b \frac{b}{a} - \epsilon_c \frac{c}{a}. \tag{3.4.5}$$

It can be observed very clearly that if both the numbers are approximately same i.e. $a \simeq b$, relative error in a becomes,

$$\frac{a_c}{a} = 1 + \epsilon_a \simeq 1 + \frac{b}{a}(\epsilon_b - \epsilon_c) \simeq 1 + \frac{b}{a}\max(|\epsilon_b|, |\epsilon_c|) \tag{3.4.6}$$

Although the relative errors in b and c are subtracted, but final relative error is multiplied by a large factor $\frac{b}{a}$ which ultimately magnifies the error.

3.4.4 Implementation and Assessment

3.4.4.1 Machine Epsilon: Listing 1 shows the code used calculate the machine epsilon of double data types. The output gives the value of machine error as $2.22045e-16$. A similar code was run to calculate the machine epsilon for float data type, which was found out to be $1.19209e-07$.

```cpp
#include<iostream>
#include <cfloat>
using namespace std;

// Function for Machine Epsilon with an
// initial value provided as EPS.
void machineEpsilon(double EPS)
{
    // taking a floating type variable
    double prev_epsilon;

    // run until condition satisfy
    while ((1+EPS) != 1)
    {
        // copying value of epsilon into previous epsilon
        prev_epsilon = EPS;

        // dividing epsilon by 2
        EPS /=2;
    }

    // print output of the program
    cout << "Machine Epsilon is : " << prev_epsilon << endl;
}
// Driver Code
int main()
{
    // calling function which calculate machine epsilon
    // with initial value provided as 0.5
    machineEpsilon(0.5);

    return 0;
}
```

Listing 1: Code to calculate the machine epsilon of double data types.

3.4.4.2 Series Sum : Subtractive cancellation is a dominant source of errors in the sum of a series containing alternative signs. Consider for an example , a finite series,

$$S_N^{(1)} = \sum_{n=1}^{2N} (-1)^n \frac{n}{n+1}. \tag{3.4.7}$$

The above sum can be divided into even and odd values of n separately, which gives,

$$S_N^{(2)} = -\sum_{n=1}^{N} \frac{2n-1}{2n} + \sum_{n=1}^{N} \frac{2n}{2n+1}. \tag{3.4.8}$$

Sum $S_N^{(2)}$ involves only subtraction of two series (without any alternative sign). Even this single subtraction might cause sufficient subtractive cancellation. Even this single subtraction can be avoided if the sum is written as

$$S_N^{(3)} = \sum_{n=1}^{N} \frac{1}{2n(2n+1)}. \tag{3.4.9}$$

Mathematically all of these sums i.e. $S_N^{(1)}$, $S_N^{(2)}$ and $S_N^{(3)}$ are equal, but they might give different numerical results. Listing 3 lists a C++ code to calculate $\log_{10}|(S_N^{(1)} - S_N^{(3)})/S_N^{(3)}|$ as a function of $\log_{10}(N)$.

```
/*
Name  - Ishwar Singh
Aim   - Error analysis in series sum.
Email - ishwarsaini10@gmail.com
*/

/*C++ Directives*/
#include<iostream>
#include<stdio.h>
#include<math.h>
#include<iomanip>
#include<fstream>


float series1(int n){
    int   i ;
    float t ;
    float s ;
    s = t = -1.0/2.0  ;
    for(i=1;  i<2*n;  i++){
        t*=-(pow((i+1),2))/(i*(i+2))  ;
        s+=t;
    }
    return(s);
}

float series2(int n){
    int   i   ;
    float t1  ;
    float t2  ;
    float s1  ;
    float s2  ;
    s1  = t1  = 1.0/2.0   ;
    s2  = t2  = 2.0/3.0   ;
    for(i=1;  i<n;  i++){
        t1*=((2*i+1)*i)/((2*n-1)*(n+1))       ;
        t2*=((2*i+2)*(2*i+1))/((2*i+3)*(2*i)) ;
        s1+=t1;
        s2+=t2;
    }
    return(s2-s1);
}

float series3(int n){
    int   i ;
    float t ;
    float s ;
    s = t = 1.0/6.0  ;
    for(i=1;  i<n;  i++){
        t*=(n*(2*n+1))/((n+1)*(2*n+3));
        s+=t;
    }
    return(s);
}
int main(){
    int N;
    float s1_final  ;
    float s2_final  ;
    float s3_final  ;
    std::ofstream myfile;
    myfile.open("series1.dat");
    std::cout <<  std::setprecision(7) << std::setw(10) ;

    for(int N=1;  N<=100; N++){
        s1_final  = series1(N);
        s2_final  = series2(N);
        s3_final  = series3(N);
        myfile <<  log10(N)  <<  "\t"  <<  log10(abs((s1_final-s3_final)/s3_final))  <<  std::endl;
    }
    myfile.close();
    return 0;
}
```

Listing 2: C++ code to calculate $\log_{10}|(S_N^{(1)} - S_N^{(3)})/S_N^{(3)}|$ as a function of $\log_{10}(N)$.

Figure 3.1 shows the variation of $\log_{10}|(S_N^{(1)} - S_N^{(3)})/S_N^{(3)}|$ as a function of $\log_{10}(N)$. The following observations can be made by analyzing this plot :

- Negative of the ordinate gives the number of significant figures used in the calculation. This variation of negative ordinate shows the loss of significance of the answer.
- There is a linear variation in the initial part of the plot which shows that the error due to subtractive cancellation is proportional to a power of N.

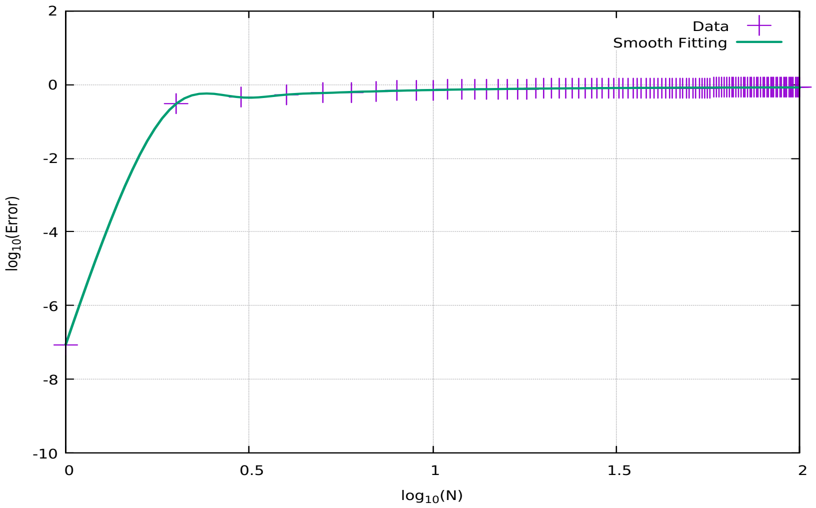

Figure 3.1: Variation of $\log_{10}|(S_N^{(1)} - S_N^{(3)})/S_N^{(3)}|$ as a function of $\log_{10}(N)$.

3.4.4.3 Errors in Spherical Bessel Functions: Spherical Bessel functions are actually solutions of the differential equation,

$$x^2 f''(x) + 2xf'(x) + [x^2 - l(l+1)]f(x) = 0. \tag{3.4.10}$$

There are various methods to calculate the values of these functions. One of the methods is the numerical recursion relations method. These recursion relations can be used both ways i.e. upward recursion relation and downward recursion relations

$$j_{l+1} = \frac{2l+1}{x} j_l(x) - j_{l-1}(x), \text{ (up)} \tag{3.4.11}$$

$$j_{l-1} = \frac{2l+1}{x} j_l(x) - j_{l+1}(x) \text{ (down)}. \tag{3.4.12}$$

```
/*
 * Name - Ishwar Singh
 * Aim  - To calculate the values and errors in Bessels functions using recursion relations.
 * Date - 08.11.2020
 * Emai - isingh@fnal.gov
*/
// C++ Directives
#include<iostream>
#include<stdio.h>
#include<math.h>
#include<cfloat>
#include<iomanip>
```

```cpp
#include<fstream>

float bessels_up (float x, int l){
    int s = 25  ;
    float a  = sin(x)/x ;
    float b  = (sin(x)/pow(x,2))-cos(x)/x ;
    float j[s]  = {a,b}  ;
    for(int k = 1;  k<s-1;  k++){
        j[k+1]  = ((2*k+1)/x)*j[k]-j[k-1];
    }
    //float scale = (sin(x)/x)/j[0];
    return j[l] ;
}

float bessels_down (float x, int l){
    int s = 25  ;
    float j[s]  = {}  ;
          j[23] = 1.0 ;
          j[24] = 1.0 ;
    for(int k = s-2;  k>0;  k--){
        j[k-1]  = ((2*k+1)/x)*j[k]-j[k+1];
    }
    float scale = (sin(x)/x)/j[0];
    return j[l]*scale ;
}
int main(){
    float x ;
    int i   ;
    std::ofstream my_file;
    my_file.open("Bessels.txt");
    for(int i =  0;  i<=25;  i++){
        x = 1.0;
        my_file <<  std::fixed  <<  std::setprecision(7)  <<  i <<  "\t"
   <<  bessels_up(x,i) <<  "\t"  <<  bessels_down(x,i) <<  "\t"  << abs(
   bessels_up(x,i)-bessels_down(x,i))/(abs(bessels_up(x,i))+abs(
   bessels_down(x,i)) <<    std::endl;
    }

    return 0;
}
```

Listing 3: C++ code to calculate the Bessel functions using upward and downward recursion relations and the associated errors.

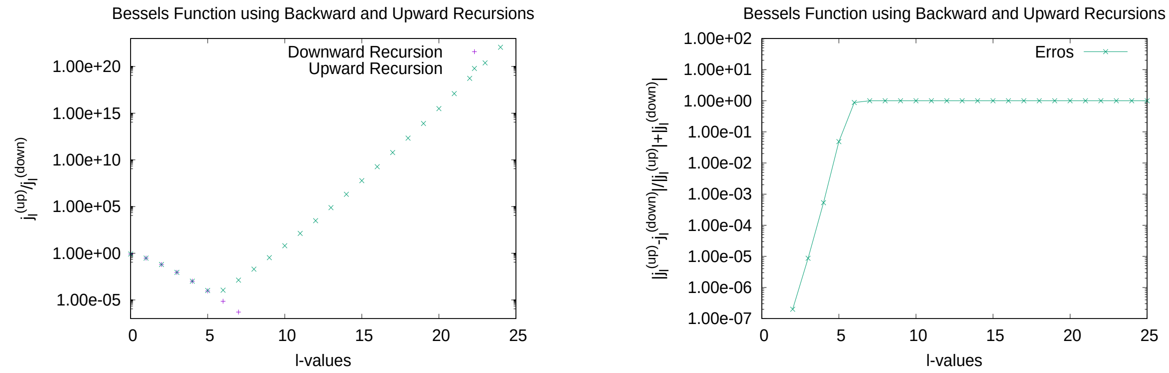

Figure 3.2: Three flavor $\nu_\mu \to \nu_\mu$ survival probabilities in both mass hierarchies in vacuum.