# ECE 2036 Lab 1: Fun with Numbers, aka Exploring Numerical Precision Errors

## *Due: Friday January 30 at 11:55 PM*

This lab is intended to help you explore various situations that can occur during numerical calculations. Please make sure that you look at the appendices in this lab as well for additional information.

You should know from ECE 2020 that a digital representation of certain real numbers is only an approximation. For example, irrational numbers such as $\pi$, $e$, $\sqrt{2}$, etc... have only a finite number of digits that can be represented in a digital computer. The two main real number representations that C++ has are single precision (32-bit) and double precision (64-bit) floating-point numbers. These variable types in C++ are called float and double, respectively. In each of the number formats, a certain number of bits is dedicated to the exponent (8 bits for single precision and 11 bits for double precision, including the exponent's sign), the mantissa (23 bits for single precision and 52 bits for double precision), and the overall sign (1 bit).

### Part 1: Computing the average of a series of numbers

In this part, you will write a C++ program to compute the average of a series of numbers and run it on the ICE cluster. The numbers will include some large numbers and some very small numbers. Your program must calculate the average in 3 different ways:

1) by first summing the large numbers, then adding the very small numbers to the sum one by one, and finally dividing the sum by the number of numbers in the series to get an average,
2) by first summing all the very small numbers, then adding the large numbers to the sum, and finally dividing the sum by the number of numbers in the series to get an average,
3) using Welford's Algorithm (discussed in class) to compute a running average of the series, which after the last number is processed will be a good approximation of the average.

Your program should generate *m* uniformly random double-precision floating point numbers between 0.0 and 100,000,000.0 and *n* uniformly random double-precision floating point numbers between 0.0000001 and 0.00000001. (For information about how to generate random numbers in C++, see Appendix D.) Compute the average of those *m*+*n* numbers using the 3 methods above. ***Make sure to use the same m+n numbers for all 3 methods!*** Your program should ask the user to input *m* and *n,* then generate the *m*+*n* random numbers, and finally output the averages computed by the 3 different methods. Use double-precision floating point for all real-valued variables in your program. Output the averages with 40 digits of precision. The following shows an example input and output:

```
Enter number of large and small numbers to average: 100  100
running average is:                  25626782.168484464287757873535156250000000000000
average with large numbers first is: 25626782.168484445661306381225585937500000000000
average with small numbers first is: 25626782.168484471738338470458984375000000000000
```

Once you have your code debugged and well tested, run it with inputs of $m = 20$ and $n =$ 1,000,000,000. Be patient while this runs – my code took over a minute to execute with these parameters on the ICE cluster! Based on the output from this execution, answer the following question.

*QUESTION 1:* Taking the running average computed via Welford's algorithm to be an accurate value, how many digits of accuracy past the decimal point do you get from each of the other two methods? Which of these two methods produced a more accurate result and why?

*Turn-in notes for Part 1:*

1. Name your program for this part `lab1Part1.cc`
2. TA will compile and test your program with the following commands from a terminal window on the ICE cluster so make sure to test your code this way before submitting it:
   - `g++ lab1Part1.cc`
   - `./a.out`

## Part 2: Approximating the value of *e*

In this part, you will write a C++ program to approximate the value of e and run it on the ICE cluster. The value of *e* can be approximated by using the following power series expansion of $e^x$:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

Written more compactly, the expansion is:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

In order to circumvent the limits of computing factorials that were discussed in the lecture, you should compute $1/n!$ from the previous term, i.e. $1/n! = (1/(n\text{-}1)!) \times (1/n)$, instead of computing it directly using a factorial function. The first term can be set manually to start this process.

Your program should approximate *e* using the power series expansion with a number of terms that is input by the user (you can assume a maximum of 100 terms if that simplifies your code). You will compare your answer to the actual value of *e* to 40 decimal places, which is:

2.7182818284590452353602874713526624977572

Your program must approximate e in two different ways:

1) using the power series expansion with $x = 1$ (this is a direct approximation of e using the power series), and
2) using the power series expansion with x = 2 (this approximates $e^2$) and taking the square root of the result to approximate e.[1]

Use double-precision floating point for all real-valued variables in your program. Output the approximations with 40 digits of precision. The following shows an example input and output:

```
Enter the number of terms to evaluate: 22

e to 40 decimal places is:          2.7182818284590450907955982984276488423347

e is approximately:                 2.7182818284590455348848081484902650117874

e^2 is approximately:               7.3890560989306450778713042382150888442993
which gives e as approximately:     2.7182818284590442026171785983024165034294
```

Once you have your code debugged and well tested, run it with a number of terms ranging from 10 to 40 in increments of 5. Plot the number of digits of accuracy past the decimal point vs. the number of terms for both the direct and the square method on the same graph. (Plotting can be done with any method you like – it does not have to be done with C++ code.)

*QUESTION 2:* How do you explain the accuracy of the two methods compared to each other as the number of terms in the power series increases?

*Turn-in notes for Part 2:*

1. Name your program for this part `lab1Part2.cc`
2. TA will compile and test your program with the following commands from a terminal window on the ICE cluster so make sure to test your code this way before submitting it:
   - `g++ lab1Part2.cc`
   - `./a.out`

[1] You can add `#include <cmath>` at the top of your source file and then use the `sqrt()` function to calculate the square root.

## Appendix A: Turn-in Instructions

1. Make a directory on ICE named `Labl`. To make this directory, type the following in a terminal while you are in your home directory:

`mkdir Labl`

2. Put your source code files (i.e. `lablPartl.cc` and `lablPart2.cc`) in this folder. To get into this folder type this at the command prompt:

`cd Labl`

3. Go back to your home directory. You can do this by typing `cd` at the command prompt.

4. From the home directory, type the following command to create a compressed tarball of your work:

```
tar -cvzf <yourusername>Labl.tar.gz ./Labl
```

5. Submit the file `<yourusername>Labl.tar.gz` via Canvas.

6. Write or type your answers to Question1 and Question 2 and turn them in via Canvas as well.

7. Also turn in your plots of accuracy vs. number of terms for the two e approximation methods.

8. Finally, turn in screenshots of your program input and output for both parts with the specified parameters for Part 1 and any parameter within the specified range for Part 2.

***Important note:*** It is your responsibility to verify your submission before turning it in to Canvas. If you follow the above instructions, you should avoid problems. However, to be absolutely certain that you are submitting your code correctly, I recommend copying your tarball to a new folder and "untarring" it to make sure your files have been included correctly. You can do this with the following commands from your home directory:

- `mkdir temp`
- `cp <yourusername>Labl.tar.gz temp`
- `cd temp`
- `tar -xvzf <yourusername>Labl.tar.gz`
- `ls Labl`
- `cat Labl/lablPartl.cc`
- `cat Labl/lablPart2.cc`

If `Labl` directory does not appear after the "untarring", there is a problem. If `Labl` directory appears but you see nothing from `ls Labl`, there is a problem. If `Labl` directory does appear, and you see your source code files displayed after these commands, you are good to submit your `<yourusername>Labl.tar.gz` file.

# Appendix B: C/C++ Basic Elements

You can look these up in the textbook for more information. We will talk in class in more depth, but I would like for you to experiment with these programming constructs.

**A. if statements in C/C++**

```
if (condition)
{
  //body of the if statement
}
```

**B. if-else statements in C/C++**

```
if (condition)
{
  //body of the if statement
}
else
{
  //body of the if statement
}
```

**C. For loop example**

```
int i;
for (i =0; i <= upperLimit; i++)  //i++ is called the increment operator that adds 1 to the value of i
{
   //body of for loop
}
```

**D. While loop example**

```
while (condition)
{
   //body of while loop
}
```

**E. Array definitions and indexing (notice zero indexing for the first element)**

```
float arrayNumbers[100]; //this is a 100 float elements array

arrayNumbers[0] = 1.5; //this is the first element in the array -- zero indexing

arrayNumbers[99]; //this is the last element in the array -- note not 100
```

# Appendix C: Good Programming Practices

## Indentation

When using *if/for/while* statements, make sure you indent 2 to 4 spaces for the content inside. For example:

```
for (int i=0; i < 10; i++)
   j = j + i;
```

If you have nested statements, you should use multiple indentations. Your *if/for/while* statement brackets *{ }* can follow two possible conventions. Each both getting their own line (like the *for* loop) OR the open bracket on the same line as the statement (like for the *if/else* statement) and closing bracket its own line. If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for (int i=0; i < 10; i++)
{
   if (i < 5) {
      counter++;
      k -= i;
   }
   else {
      k += i;
   }
   j += i;
}
```

## Camel Case

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. firstSecondThird). This applies for functions and member functions as well! The main exception to this is class names, where the first letter should also be capitalized.

## Variable and Function Names

Your variable and function names should be clear about what that variable or function is. Do not use one letter variables, but use abbreviations when it is appropriate (for example: “imag" instead of “imaginary”). The more descriptive your variable and function names are, the more readable your code will be. This is one of the ideas behind self-documenting code.

## Clear Comments

Some good opportunities to use comments are…

- Introducing a member function or class
- Introducing a section of code with long implementation
- Your name, class information, etc. at the beginning of the file

## Appendix D: Random Numbers in C++

C++ has a library named "random", which includes several different random number generation algorithms and supports a wide variety of probability distributions. For a complete specification of the random library, see:

https://cplusplus.com/reference/random/

For this lab, you will use the "uniform_real_distribution" class from the random library. You can find several examples that generate uniformly distributed real numbers at:

https://www.geeksforgeeks.org/cpp/stduniform_real_-distribution-class-in-c-with-examples/

I recommend using the Mersenne Twister (mt19937), which is a very good algorithm for random number generation, as the generator for your uniform real distributions. An mt19937 object named `generator` is instantiated with the following C++ statement:

```
mt19937 generator(seed);
```

where `seed` is an integer value used to initialize the pseudorandom number generator. Using a fixed seed such as '1' will give you the same sequence of random numbers every time you run the program. This is sometimes useful for debugging since it produces the same behavior every time. If you want to generate different random behavior every time you run the program you can seed the random generator with the system time as follows:

```
mt19937 generator(time(NULL));
```

## Appendix E: ECE 2036 Lab Grading Rubric

If a student's program runs correctly and produces the desired output, the student has the potential to get a 100 on the lab; however, other elements are necessary to meet the lab requirements, as indicated below. In addition, if a student's code does not compile, there is an automatic deduction on the lab (20% for each part that does not compile). Code that compiles but does not match the sample output can incur a deduction from 10% to 30% depending on how poorly the output matches the output specified by the lab. This is in addition to the other deductions listed below or due to the student not attempting the entire assignment.

**AUTOMATIC GRADING POINT DEDUCTIONS**

| Element | Percentage Deduction | Details |
|---|---|---|
| Does Not Compile | 40% | Programs do not compile on ICE cluster! (20% per part) |
| Does Not Match Output | 10%-30% | The programs compile but outputs are incorrect |

**ADDITIONAL GRADING POINT DEDUCTIONS**

| Element | Percentage Deduction | Details |
|---|---|---|
| Clear Self-Documenting Coding Styles | 5%-15% | This can include incorrect indentation, using unclear variable names, not enough or unclear comments, etc. |
| Answers to questions | 20% | Unsatisfactory answers (10% each), partial credit possible |

**LATE POLICY**

| Element | Percentage Deduction | Details |
|---|---|---|
| Late Deduction | See details | < 2 hours late: 5 points off total<br>> 2 hours late, up to 1 day late: 10 points off total<br>> 1 day late, up to 2 days late: 20 points off total<br>20 additional points off for each day late beyond 2 days |