# ECE 2036 Lab 1: Exploring Numerical Error

***Due:*** Friday Sept 7, 2018 at 11:59 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.

## PART 1: OUT OF RANGE ERROR

In C++, the fundamental type of `int` on the PACE-ICE system uses 32 bits to represent an integer; however, you can extend this to 64 bits by using another fundamental type in C++ called `long int`. (Please note that these sizes can vary from platform to platform and are not guaranteed in the C++ standard which we will talk about in class.) I would like for you to write a program to calculate the factorial using first the `int` type and then the `long int` type. Please empirically show how large of a factorial you can hold in an `int` variable and a `long int` variable. In addition to the `main()` function, to get practice using global functions, I would like for you to create two global functions to calculate the factorial. These two global functions should have one argument that is an int. The first should return an `int`, and the second should return a `long int` so that you can compare the ranges of these two data types. In your Lab1 folder, please call your source code `lab1part1.cc`. Sample output for this program up to 5! is shown below. Please follow this output format; however, I would like for you to show calculations up to 25!

```
Factorial results using int
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120

Factorial results using long int
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

## PART 2: ROUND-OFF ERROR

You should know from ECE2020 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 representations that we will use in C++ will be single precision (32-bit) and double precision (64-bits) numbers. 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), the mantissa (23 bits for single precision and 52 bits for double precision), and 1 bit for the sign bit.

I would like for you to use a C++ program to calculate the roots of the following quadratic equation:

$$x^2 + 3000.001x + 3 = 0$$

Please use the following conventional quadratic equation in your program to solve for the roots.

$$x1, x2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In fact, I would like for you to use a global function that has four parameters that you pass to it (i.e. a flag to indicate if it should return the plus or minus solution, a, b, and c). You should make one function using only `float`'s and the other only `double`'s so that you can compare the results. To calculate the square root, you can use the `sqrt(x)` function that is a part of the `cmath` library. To include this in your program you must have the following preprocessor directive.

```
#include <cmath>
```

The actual exact roots of this equation are $x_1 = -0.001$ and $x_2 = -3000$. Please compare how your program calculates this with first `float` types and then with `doubles`. You can calculate the error given by:

$$\% \text{ error} = 100*(\text{actual} - \text{approximation})/\text{actual}$$

*QUESTION YOU NEED TO ANSWER:* Given the fact that both roots can easily be represented with a `float`, why do you get a couple percent error for one of the roots?

In your Lab1 directory, please call your source code `lab1part2.cc`. Possible sample output is as follows.

```
Using the float data type the roots are:
x1 = <you calculate>     % error = <you calculate>
x2 = <you calculate>     % error = <you calculate>

Using the double data type the roots are:
x1 = <you calculate>     % error = <you calculate>
x2 = <you calculate>     % error = <you calculate>
```

## PART 3: TRUNCATION ERROR

In numerical analysis, the term "truncation error" is typically related to the error introduced by not using all the terms in a series expansion or a limited number of iterations used in approximations to differential equation solutions. To illustrate this, I would like for you to

calculate the truncation error in calculating the value *e* 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 gives the expansion as:

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

Given what you have learned from the previous parts of this lab, I would like for you to write a global function that calculates each series term using the PREVIOUS series term of the expansion. This strategy will enable you to circumvent limits explored in Part 1.

Please use your program to determine how many terms in the power series are needed to use the full precision of a `float` and then of a `double` (i.e. when the number values stop changing). You will compare your answer to the actual value of *e* given 50 decimal places:

*e = 2.71828182845904523536028747135266249775724709369995*

For this experiment, you can shove this in a global double variable.

`double e = 2.71828182845904523536028747135266249775724709369995`

In your Lab1 directory, please call your source code `lab1part3.cc`. An example of the output of your program might look like:

```
Input the number of terms in the power series using FLOATS: 4
--------
#terms = 1
approx = 1
%error = 63.212
--------
#terms = 2
approx = 2
%error = 26.424
--------
#terms = 3
approx = 2.5
%error = 8.0301
--------
#terms = 4
approx = 2.666666746139526367l875
%error = 1.8988

Input the number of terms in the power series using DOUBLES: 4
--------
#terms = 1
```

```
approx = 1
%error = 63.212
--------
#terms = 2
approx = 2
%error = 26.424
--------
#terms = 3
approx = 2.5
%error = 8.0301
--------
#terms = 4
approx = 2.66666666666666651863693004997912794351577758789062 5
%error = 1.8988
```

## PART 4: ERROR PUZZLE

I would like you to create code to calculate *e* using 100 terms of the power series expansion. You will compare your answer to the actual value of *e* given 50 decimal places as mention in part 3.

However, there is a little bit of a twist in this last part. First, I would like you to calculate a cumulative sum in order from the largest term to the smallest term in the power series. Then, I would then like for you to sum the values in the reverse order. You might consider putting each term in the power series in an array with 100 elements. You can then use a sum term that starts summing from the beginning of the array for the forward summation. You can compare this answer to the sum starting at the end of the array.

In theory you would think it should not make a difference which order you sum the values, but you should see a difference in the result. Please make sure your program shows the difference between the two. Also, please do this with `float` types and then `double` types.

Why do you think there is a difference between these the forward and backward summations?

In your Lab1 directory, please call your source code `lab1part4.cc`. Sample output of your program should look like:

```
--------
forward approx = <your value>
forward %error = <your value>
backward approx = <your value>
backward %error = <your value>
--------
forward approx = <your value>
forward %error = <your value>
backward approx = <your value>
backward %error = <your value>
```

## APPENDIX A - Creating Lab1 Program on PACE-ICE and Submission Instructions

**Creating Source Code.** Once logged into PACE-ICE, you will need to use one of several available text editors found on the PACE-ICE cluster. Those are vi/vim, nano and emacs. I will use vi in class, and I will recommend that you become familiar with this editor. There are many online tutorials to help you get used to vi. You could get started here.

https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

**Basic Unix Commands.** Please make a directory (i.e. "folder") called `Lab1` where you keep your files. To make this directory use the following command in your home directory:

```
mkdir Lab1
```

To go into this directory from your home directory, you can use the following command:

```
cd Lab1
```

To get back to your home directory you can use the command:

```
cd
```

**Compiling Source Code.** On the PACE-ICE system we will be using the gnu g++ and gcc compiler. Use the following command to compile your source code and create an executable file called `testProgram`.

```
g++ testProgram.cc -o testProgram
```

To run your program at the command prompt, type in the executable file name

```
./testProgram
```

**Submitting Assignments.** You will need to submit BOTH your SOURCE CODE AND THE TURN IN SHEET ON CANVAS. You will need to download the SOURCE code from PACE-ICE to your local machine, and then upload to the Canvas assignment.

1. I would like for you to make a directory on PACE-ICE called `Lab1`. To make this directory, type the following while you are in your home directory.

```
mkdir Lab1
```

2. Please make all your source code files (i.e. `lab1part1.cc`, `lab1part2.cc`, `lab1part3.cc`, and `lab1part4.cc`) in this folder. To get into this folder you can use at the command prompt:

```
cd Lab1
```

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

4. At the home directory, please type the following command to create a compressed `tarball` of your work

```
tar -cvzf yourusernameLab1.tar.gz ./Lab1
```

5. As I showed you in class you will need to download this file to your local machine. You will then submit this yourusernameLab1.tar.gz to Canvas so that the TAs can grade it.

6. In Appendix C, there is a turn-in sheet. Please make sure that you fill this out and upload this to Canvas as well.

## APPENDIX B - C/C++ Basic Syntax

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; //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 - Turn-in Sheet

**Problem 1: OUT OF RANGE ERROR**

Please turn in a program that shows what the largest factorial number you can exactly calculate.

The largest factorial that can be represented with regular `int` type is 33.

The largest factorial that can be represented with a `long int` type is 65.

**Problem 2: ROUND-OFF ERROR**

Please turn in the program you wrote to calculate the values in the following chart.

Using a `float`

x1 = -0.001 %error from actual root = -0.0000047497

x2 = -3000 %error from actual root = -0

Using a `double`

x1 = -0.001 %error from actual root = 0.0000000024

x2 = -3000 %error from actual root = -0

Given the fact that each root can easily be represented in a float. Why do you think that there was error using the floating point? Be as specific as you can. You might write your answer on the back of this sheet.

**Problem 3: TRUNCATION ERROR**

How many terms do you need to include in the power series expansion until the digital value remains unchanged?

answer for `float` type: 11

answer for `double` type: 18

**Problem 4: ERROR PUZZLE**

What is the error for the summing the first 100 terms in the power series from largest to smallest.

answer for `float` type: 2.718281984329223632 8125

answer for `double` type: 2.7182818284590455348848081484902650117874145507 8125

What is the error for the summing the first 100 terms in the power series from smallest to largest.

answer for `float` type: 2.7182817459106445312 5

answer for `double` type: 2.7182818284590450907955982984276488423347473144531 2

Why do you think it makes a difference as to whether you sum forward or backwards?

ANS: I am not sure. I am guessing that it is because of the way the program adds different values using double or float.

# APPENDIX D: ECE2036 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 his or her lab; however, TA's will **randomly** look through this set of "perfect-output" programs to look for other elements of meeting the lab requirements. The table below shows typical deductions that could occur.

In addition, if a student's code does not compile, then he or she will have an automatic 40% deduction on the lab. Code that compiles, but does not match the sample output can incur additional deductions 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 PACE-ICE! (10% for each part) |
| Does Not Match Output | 10%-30% | The programs compile but don't match all output |
| Answers on Turn-In Sheet in Appendix C | 30% | Make sure you fill the turn-in sheet and post this to canvas. |

**ADDITIONAL GRADING POINT DEDUCTIONS FOR RANDOMLY SELECTED PROGRAMS**

| Element | Percentage Deduction | Details |
|---|---|---|
| Global Functions | 10% | Does not use any global functions. |
| Clear Self-Documenting Coding Styles | 5%-15% | This can include incorrect indentation, using unclear variable names, unclear comments, or compiling with warnings. (See Appendix E) |

**LATE POLICY**

| Element | Percentage Deduction | Details |
|---|---|---|
| Late Deduction Function | score - (20/24)*H | H = number of hours (ceiling function) passed deadline<br>note : Sat/Sun count has one day; therefore H = 0.5*$H_{weekend}$ |

# Appendix E: Good Programming Practices

*Indentation*

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

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

If you have nested statements, you should use multiple indentions. 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; i < 10; i++)
{
  if (i < 5) {
    counter++;
    k -= i;
  }
  else {
    k += i;
  }
  j += i;
}
```

*Camel Case (Suggested But Not Required)*

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 the idea 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 F: Accessing PACE-ICE Instructions

## *ACCESSING LINUX PACE-ICE CLUSTER (SERVER)*

To access the PACE-ICE cluster you need certain software on your laptop or desktop system, as described below.

### Windows Users:

Option 0 (Using SecureCRT)- THIS IS THE EASIEST OPTION!

The Georgia Tech Office of Information Technology (*OIT)* maintains a web page of software that can be downloaded and installed by students and faculty. That web page is:

http://software.oit.gatech.edu

From that page you will need to install SecureCRT.

To access this software, you will first have to log in with your Georgia Tech user name and password, then answer a series of questions regarding export controls.

Connecting using SecureCRT should be easy.

- Open SecureCRT, you'll be presented with the "Quick Connect" screen.
- Choose protocol "ssh2".
- Enter the name of the CoC machine you wish to connect to in the "HostName" box (i.e. *coc-ice.pace.gatech.edu*)
- Type your username in the "Username" box.
- Click "Connect".
- A new window will open, and you'll be prompted for your password.

Option 1 (Using Ubuntu for Windows 10):

Option 1 uses the Ubuntu on Windows program. This can only be downloaded if you are running Windows 10 or above. If using Windows 8 or below, use Options 2 or 3. It also requires the use of simple bash commands.

1. Install Ubuntu for Windows 10 by following the guide from the following link:

   https://msdn.microsoft.com/en-us/commandline/wsl/install-win10

2. Once Ubuntu for Windows 10 is installed, open it and type the following into the command line:

   *ssh **YourGTUsername**@ coc-ice.pace.gatech.edu*

   where **YourGTUsername** is replaced with your alphanumeric GT login. Ex: bkim334

3. When it asks if you're sure you want to connect, type in:
   *yes*

and type in your password when prompted (Note: When typing in your password, it will not show any characters typing)

4. You're now connected to PACE-ICE. You can edit files using vim by typing in:

   *vi filename.cc* OR *nano vilename.cpp*

   For a list of vim commands, use the following link:

   https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

5. You're able to edit, compile, run, and submit your code from this command line.

Option 2 (Using PuTTY):

Option 2 uses a program called PuTTY to ssh into the PACE-ICE cluster. It is easier to set up, but doesn't allow you to access any local files from the command line. It also requires the use of simple bash commands.

1. Download and install PuTTY from the following link:
   www.putty.org

2. Once installed, open PuTTY and for the Host Name, type in:
   *coc-ice.pace.gatech.edu*

   and for the port, leave it as 22.

3. Click Open and a window will pop up asking if you trust the host. Click Yes and it will then ask you for your username and password. (Note: When typing in your password, it will not show any characters typing)

4. You're now connected to PACE-ICE. You can edit files using vim by typing in:
   *vim filename.cc* OR *nano vilename.cpp*

   For a list of vim commands, use the following link:

   https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

5. You're able to edit, compile, run, and submit your code from this command line.

**MacOS Users:**.

Option 0 (Using the Terminal to SSH into PACE-ICE):

This option uses the built-in terminal in MacOS to ssh into PACE-ICE and use a command line text editor to edit your code.

1. Open Terminal from the Launchpad or Spotlight Search.

2. Once you're in the terminal, ssh into PACE-ICE by typing:

   *ssh **YourGTUsername**@ coc-ice.pace.gatech.edu*

   where **YourGTUsername** is replaced with your alphanumeric GT login. Ex: bkim334

3. When it asks if you're sure you want to connect, type in:
   *yes*

   and type in your password when prompted (Note: When typing in your password, it will not show any characters typing)

4. You're now connected to PACE-ICE. You can edit files using vim by typing in:

   *vi filename.cc* OR *nano filename.cpp*

   For a list of vim commands, use the following link:

   https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started

5. You're able to edit, compile, run, and submit your code from this command line.

**Linux Users:**

If you're using Linux, follow Option 0 for MacOS users.