Floating-Point Arithmetic Exercises and Examples

We will go through several examples of floating-point code which demonstrates some of the points made during the lectures

- One of the openlab staff will describe which system you should use and in which directory you will find the exercises. Set your working directory to that location.
- The code in the exercises can be compiled with gcc 4.8 and icc 13.0 update #2.
- **Source** the setup script `Setup-for-exercises` and then verify that the correct compilers are available to you. You should see something like

```
-bash-4.1$ . ./Setup-for-exercises
-bash-4.1$ which gcc && gcc -v
/oplashare/sw/linux/x86_64/gcc/slc6/gcc-4.8.0/bin/gcc
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/oplashare/sw/linux/x86_64/gcc/slc6/gcc-
4.8.0/libexec/gcc/x86_64-unknown-linux-gnu/4.8.0/lto-wrapper
Target: x86_64-unknown-linux-gnu
Configured with: ../gcc-4.8.0/configure --enable-languages=c,c++,fortran
--prefix=/oplashare/sw/linux/x86_64/gcc/slc6/gcc-4.8.0
Thread model: posix
gcc version 4.8.0 (GCC)
-bash-4.1$ which icc && icc -v
/oplashare/sw/linux/x86_64/intel/xe2013/composer_xe_2013.2.146/bin/intel64/ic
c
icc version 13.1.0 (gcc version 4.7.0 compatibility)
```

- In those exercises which supply a makefile, define `CXX=icc` to use icc to build the project instead of the default compiler gcc. The default target is `all`: build all the files required in the exercise. There is also a `realclean` target to delete all the files created. NOTE: you must build the `realclean` target before switching compilers. A typical sequence might be

```
make # build with gcc
# work on the exercise
# save any files which you way want later; e.g., executables
make realclean # clean before switching compilers
make CXX=icc # build using icc
# work on the exercise
# save any files which you way want later
make CXX=icc realclean # remove icc-created files
```

01_numeric_limits

This program demonstrates use of the templated **`numeric_limits`** functions. The program prints the values of various significant floating-point quantities relevant to **`float`**, **`double`**, **`long double`** and **`__float128`** datatypes. Note that **`__float128`** is not consider a fundamental datatype by the GNU compiler system. Thus, the standard GNUJ C++ headers do not include instantiations specialized for **`__float128`**. They are supplied in **`numeric_limits.hpp`**. They may not be entirely correct.

Compare the outputs of the gcc and icc verisons. They should be the same except for a few **`long double`** exceptional values. The differences appear to be the result of how "ignored" bits in the 80-bit storage format are handled by the two compilers. Note that the **`long double`** routines **are** part of the standard C++ header **`limits`**.

02_decode

This sample program decodes the sign, exponent and significand fields of the **`single`**, **`double`**, **`long double`** and **`quad`** floating-point data types. It demonstrates the use of C language system header files to map the various fields of the storage format of a floating-point number onto structures which can be used to examine (or modify) them.

03_summation

This directory contains the files to build a program which sums a collection of double precision values in a number of different ways. The program also measures the execution time of each method.

The summation techniques used include

- simple summation, with and without sorting
- summation using increased precision
- summation using a high precision math library (mpfr)
- an error free transformation (EFT) which implements a highly accurate summation scheme without use of increased precision
- vectorization
- unrolled loops
- OpenMP
- reduction methods from Intel® Threading Building Blocks (TBB)

First make some observations:

- Run the program several times after building it with both gcc and icc.
- Are the results always the same? Are they the same for both compilers?
- Which techniques give the same result each time?
- Which give results which vary?
- Can you explain why?

- Examine the manner in which the pseudo-random numbers are generated. They aren't randomly distributed on $[0,1)$.
- Look at the source files containing the various functions. Understand how the various techniques are implemented.
- The main program demonstrates a method to measure elapsed time using OpenMP. Note that OpenMP is only used "computationally" in one of the summing routines.

Which of the techniques do you think provides the "right" value for the sum. What do you think "right" means in this case?

Examine the way in which the pseudo-random numbers are generated. Can you correlate that with the different results? E.g., compare the "unroll by 4" results with the "unroll by 5" results.

Identify the techniques which use multiple threads of execution or multiple partial sums. Why does this affect the results?

04_dot_product

This directory contains several prototype programs for calculating dot products. Examine each one and see how the various error free transformations discussed during the lecture are applied to the problem.

If you wish to be creative, create a main program which incorporates one or more of these routines and use it to calculate a dot product. Can you correlate your results with the condition number associated with the dot product?

05_quadradic

This directory contains a simple program which solves the equation

$$f(x) = ax^2 + bx + c$$

for its real roots. It does so in a very numerically naïve way.

Compile the program and fill in this table (where x_+ and x_- are the roots found) using results from the naïve version:

a	b	c	x_+	x_-	$f(x_+)$	$f(x_-)$
+1	+2000	-3				
+2	-4000	-1				
+5	+8000	+2				

You may wish to calculate the values of $f(x_+)$ and $f(x_-)$ by hand or with a calculator. Verify your results.

Study the values of the roots as displayed by the program. You should notice that the two roots are very different in magnitude; the ratio of their magnitudes is $\sim 10^6$. This is usually an indication of an ill-conditioned problem. Notice also that the low-order hex digits of the smaller root are usually repeated digits, often 0. This is caused by catastrophic cancellation in the calculation.

Now try the "improved" version of the program. (It is created by compiling the same source file with **–D_IMPROVED**.) Notice that the smaller root is now calculated more accurately and that the value of $f(x_{smaller})$ is closer to 0.

Solving the quadratic equation more accurately provides a simple example of how catastrophic cancellation can be removed from a problem algebraically.

The roots are given by

$$\begin{aligned} x_\pm &= \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \\ &= -\frac{b}{2a}\left(1 \mp \sqrt{1 - \frac{4ac}{b^2}}\right) \end{aligned}$$

Let $\delta = 4ac/b^2$. Then

$$x_+ = -\frac{b}{2a}\left(1 - \sqrt{1-\delta}\right)$$

When $b^2 \gg 4ac$ (i.e., $\delta \ll 1$), the calculation of x_+ involves the taking the difference of two nearly equal quantities, resulting in catastrophic cancellation.

We can rationalize the expression for x_+ by multiplying numerator and denominator by $1 + \sqrt{1-\delta}$ giving

$$\begin{aligned} x_+ &= -\frac{b}{2a}\left(\frac{1-(1-\delta)}{1+\sqrt{1-\delta}}\right) \\ &= -\frac{2c}{b}\left(\frac{1}{1+\sqrt{1-\delta}}\right) \end{aligned}$$

and

$$x_- = -\frac{b}{2a}\left(1 + \sqrt{1-\delta}\right)$$

Now there is no catastrophic cancellation when x_+ is computed if $b^2 \gg 4ac$.

06_sollya

Sollya is a tool environment for developing floating-point code. Complete documentation is available via the web site.

In this directory, you will find a simple sollya script which investigates the 0.1/0.01 approximation situation. Run the script and observe the output. Examine the script to see how sollya is used.

See if you can modify the script to study another $x/2 * x$ type problem. E.g., choose 0.2 and 0.04. Does the product **`0.2*0.2`** match **`0.04`** when everything is computed in double precision?