|epcc|


# Numerical Computing Practical

## 1 Introduction

This practical gives a short introduction to the difficulties we encounter when trying to represent infinite precision real numbers as finite precision floating point numbers on computers.

## 2 Compilers

For these exercises, you should use the following compilers:

`gcc` for C;

`g++` for C++;

`gfortran` for Fortran;

`javac` for Java.

Any compiler options mentioned in the rest of the text are specific to these compilers. Other compilers will typically have similar options, but you'll have to read the manual to find out what they are.

# 3 Limited Range and Precision

Write a simple program (in any of the languages mentioned above) that does the following.

1. Set $N = 20$
2. Set $a = 2$
3. loop for $i = 1$ to $N$
    - print $a$
    - $b = a \times 2$
    - print $b$
    - $c = b + 2$
    - print $c$
    - $d = c/2$
    - print $d$
    - $a = a \times 10$

If we had infinite precision, we would always find that this program would print $d = a + 1$. The point of this exercise is to investigate what answers we obtain in practice using different types of variables.

You should do this calculation three times:with $a$, $b$, $c$ and $d$ declared as

1. integers

2. single precision floating point

   C, Java, C++ float
   Fortran real(kind=4)

3. double precision floating point

   C, Java, C++ double
   Fortran real(kind=8)

Think about these questions:

1. When and why does the integer calculation give incorrect answers?

2. When and why does the single precision calculation give incorrect answers?

3. Do the results agree with the ranges and precisions stated in the lectures?

Now increase the value of $N$ to 500 and rerun the three programs. Are the results what you would expect?

# 4 Heron's Formula

Given a triangle for which we only know the lengths of the sides ($a$, $b$ and $c$), it is possible to calculate the area $A$ directly. The "textbook" approach is to use Heron's formula,

$$A = \sqrt{s(s-a)(s-b)(s-c)} \tag{1}$$

where

$$s = \frac{a+b+c}{2}.$$

Unfortunately, this equation performs badly numerically when the triangle is very slender. This can avoided by using a rearranged (but algebraically identical) version of Heron's formula. Order the side lengths such that $a \geq b \geq c$ then:

$$A = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4} \quad (2)$$

where

$$a \geq b \geq c.$$

Note that when implementing this, all brackets should be left in, to ensure that the order of additions and subtractions is correct.

## 4.1 A few triangles

Implement both these formulae using single and double precision floating point routines. Then calculate the area of a triangle where

- $a = 12345679.0$
- $b = 12345678.0$
- $c$ takes on the values given in the following table:

| $c$ | Area (32-bit, eqn. 1) | Area (64-bit, eqn. 1) | Area (32-bit, eqn. 2) | A |
|---|---|---|---|---|
| 4.0 | | | | |
| 2.0 | | | | |
| 1.1 | | | | |
| 1.01 | | | | |

Assuming that the final column contains the most reliable results, what do you conclude from the values obtained with the other three methods?

## 4.2 Rounding modes

The variations in the results obtained above come from rounding errors. As a result, we would expect that unreliable results might change if the rounding mode were changed. A reliable result will not change significantly since a good algorithm should not be sensitive to such small variations.

Read the manual page for your compiler (`man compiler-name`) and check to see if rounding modes can be changed at compile time. How do the values in the table change with different rounding modes, and is this what you'd expect?

# 5 Floating-point exceptions

The behaviour on encountering a floating point exception can often be changed at compile time. For example, you may be able to choose which errors to trap (that is, halt execution) or which to ignore. Again, look at the man page for your compiler to see if this is possible. Re-run the exercise on limited range and precision

to see if you can trap the inexact errors. Additionally, run with a large value of $N$ with all trapping turned off: do you understand the output?

# 6 Random Numbers

In this exercise you get to call a random number generator multiple times to produce a sequence of random numbers. The random number generator you will use should produce an approximately uniform distribution of real numbers in the range $[0, 1)$ (i.e. greater than or equal to 0 but less than 1), and is "seeded' which allows the sequences of random numbers to be reproducible. In addition, the function you will use, `uni()`, gives the same results whether called from C or Fortran.

The given code presently does nothing other than call the random number generator once to give a single random number which is then printed out.

1. You should compile and run the code a few times, adding a few more calls to the random number generator within the code, to be sure you're happy that a sequence of different random numbers is produced and that for a given seed value this sequence is reproducible.

2. Alter the code so that the random number generator is called in a loop (once per loop iteration) to produce a sequence of random numbers of a chosen length (say 100 to start with). It will be useful for the next action if you store these in an array. Print these out and again run it a couple of times to check the sequence is the same each time (a quick scan should be enough to see if anything is going wrong).

3. Find the average of this set of random numbers. Is it what you expect? Try summing both up and down the sequence. Make sure it is the same sequence both times (which is why it's useful to have these in an array). What happens when you increase the number of random numbers to, say 1000 or 10000? You may want to supress the printing of the random numbers themselves!

4. The next thing you should check is that the distribution across the full range, $[0, 1)$, really is approximately uniform. The easiest way to test this is to produce a histogram by splitting the range $[0, 1)$ into equally-sized "bins" and keeping a count of how many random numbers fall into each bin. Start with 20 equal-sized bins and try with 10000 random numbers, printing out how many fall into each bin. Does this give a fairly even distribution?

5. One of the consequences of the *Central Limit Theorem* is that despite the fact that the distribution of random numbers across the range $[0, 1)$ here should be uniform, if sample groups of random numbers are taken from this distribution and averaged then the distribution of these averages should give a *normal* distribution (look this up to see an example!).

   To test this out, keep the above loop but add an inner loop which averages short sequences of random numbers (say 10 in each sequence). In order to look at the distribution of these averages you will need to produce a histogram again (using the code you already have above).

   Does this give a normal distribution? What happens if you change the loop lengths, e.g. the inner loop to 1, or something relatively large? What happens if you change the bin sizes?