Workshop 5: Floating point

In this workshop we will look at floating point accuracy and floating point exceptions. Section 2 looks at how to estimate the accuracy of a `float` and a `double` (based on the method from the "Determining Accuracy" section in chapter 16 of "Practical C Programming"). Section 3 looks at numerical errors vs. round-off errors when using the power series to calculate a sine function (also based on chapter 16 of "Practical C Programming"). Finally in section 4 we will look at how to find out where a floating point exception is being generated in a program.

The workshop is designed to be carried out using the Linux PCs in the Lovelace laboratory. It is assumed that you are not on campus and will access these computers remotely. On the module ELE page there are instructions describing how to log in to a remote Linux computer under the "Workshop 2" section.

1 Getting set up

- Start by logging in to a Lovelace lab Linux PC, using the Linux computer availability dashboard (`http://students.emps.ex.ac.uk/dashboard/`) to choose a computer where the system load is low (you may need to use the VPN to view the dashboard).
- The example programs for this workshop are provided as a tar file. Take a copy of the tar file for this workshop by running the command:

  ```
  cp /secamfs/userspace/ug/shared/ecm3446/workshop5.tar .
  ```

  where the dot at the end of the command causes the file to be copied into the current working directory.
- Unpack the tar file by running the following command:

  ```
  tar xvf workshop5.tar
  ```

  This will create a directory called `workshop5` which contains the example programs for this workshop.

2 Determining accuracy

We can estimate the accuracy of a floating point representation by adding increasingly small numbers to 1.0 and working out when the result is indistinguishable from 1.0. For example we start by calculating $1.0 + 0.1$ and see if the result is distinguishable from 1.0. If the result is distinguishable from 1.0 then we calculate $1.0 + 0.01$, and so on until we obtain a result which is indistinguishable from 1.0.

The example program `float.c` from "Practical C Programming" implements this test in two ways. Firstly the number to be compared with 1.0 is calculated and compared directly with 1.0. Secondly the number to be compared with 1.0 is calculated and written out to memory before being read back in and compared with 1.0. This is done to test whether there is any change in accuracy when numbers are written out to memory.

- The program `float.c` and a makefile for building it can be found in the `accuracy` directory. Start by building and running the `float.c` program to measure the accuracy of a float:

```
cd workshop5/accuracy
make float
./float
```

Question 1: How many digits of accuracy does a float provide?

- There is a second version of the program in the same directory (called `double.c`) which uses doubles in instead of floats. Next build and run `double.c` to measure the accuracy of a double:

```
make double
./double
```

Question 2: How many digits of accuracy does a double provide and is this consistent with IEEE754 double precision?

3 Numerical error vs. round-off error

The function $\sin(x)$ can be calculated using the following power series

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \tag{1}$$

Terms can be added until the solution converges to the required accuracy. The program presented in Appendix D of "Practical C Programming" calculates a value of $\sin(x)$ by computing this series using a reduced precision floating point format. The floating point format has 4 digits in the significand and uses a base of 10. Terms are computed until the result is accurate to within the floating point round-off. Using a reduced precision floating point format allows us to more easily explore the balance between numerical errors and round-off in this power series calculation. Practical implementations of the sine function are more complex than just a power series but it will be informative to experiment with this simpler method.

- Change into the `sine` directory, build the program and run the program to calculate $\sin(0.5)$

```
cd ../sine
make sine
./sine 0.5
```

Question 3: How many terms are required in the power series to compute $sin(0.5)$ to within the round-off error of the reduced precision floating point format?

- In the `sine` directory there is a shell script called `run_sine.sh`. This script runs the sine program multiple times to calculate $\sin(x)$ for values of x between -3.0 and 3.0 (in increments of 0.4) and reports the number of terms calculated. Run the script (using the command `./run_sine.sh`) and note how many terms are calculated for each value of x.

Question 4: How does the number of terms required for convergence vary with the argument of the sine function (x)?

4 Floating point exception

In this last section we will see how to find out where a floating point exception is being generated in a program. The Fourier transform of a top-hat (or box car) function arises in signal processing and optics. The result of the Fourier transform of a top hat function is the function

$$y = \frac{\sin(x)}{x} \tag{2}$$

This is known as a sinc function ("sinc" with a "c" instead of "sine" an "e") and it is plotted in figure 1. The example program in this section calculates values of the sinc function.

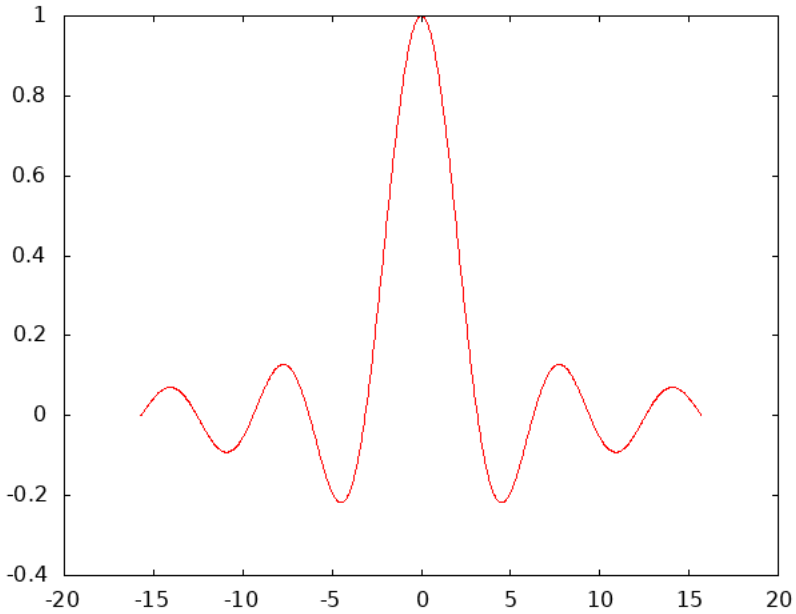

Figure 1: The sinc function: $y = \frac{\sin(x)}{x}$

- Start by changing into the `exception` directory, then compile and run the `sinc.c` program

  ```
  cd ../exception
  gcc -o sinc sinc.c -lm
  ./sinc
  ```

  The program will report that a `NaN` has been generated! The program uses a function called `isnan` to check whether the value of `y` is a `NaN` (there is a similar function called `isinf` which checks for infinities).

  Question 5: Why is a `NaN` being generated and how would you modify the program to stop this happening?