Exercises for the lecture

Fundamentals of Simulation Methods

WS 2015/16

Lecturer: Volker Springel

Exercise sheet 1 ***(due date: Oct 23, 2015, 11am)***

Issues of floating point arithmetic

1) Machine epsilon

Write a computer program in C, C++, or Python[1] that experimentally determines the machine epsilon ϵ_m, i.e. the smallest number ϵ_m such $1 + \epsilon_m$ still evaluates to something different from 1, for the following data types:

(a) `float`

(b) `double`

(c) `long double`

2) Pitfalls of floating point arithmetic

Consider the following numbers:

```
double  a =  1.0e17;
double  b = -1.0e17;
double  c = 1.0;
double  x = (a +  b) + c;
double  y =  a + (b  + c);
```

Calculate the results for x and y. Which one is correct, if any? Explain, why the law of associativity is here broken.

3) Pitfalls of floating point representation

Consider the following C/C++ code:

[1]Special note for Python: Floating point numbers in python have a fixed default precision. In order to enforce a precision like you do in C/C++, you have to use the numpy module. You can create a 32 bit variable like this:
`a = np.float32(2.)`
You have to encapsulate numerical constants in your operations as well:
`b = a * np.float32(1.5)`
You can check the type of a variable with "`type(a)`" (This should give "`numpy.float32`").
Bonus question for python users: What is the default machine precision of python floats?

```
float   x =  0.01;
double  y =  x;
double  z  = 0.01;

int     i = x * 10000;
int     j = y * 10000;
int     k = z * 10000;

printf("%d  %d  %d\n", i, j, k);
```

which prints out three integer numbers.

(a) Explain why the numbers are not all equal.

(b) Determine the rational number n/m, where n and m are natural numbers, that is represented by the single-precision IEEE-754 floating point variable `x` in the above example. Note: This number is not 1/100.

4) Packing of numbers

Estimate how many numbers there are in the interval between 1.0 and 2.0, and in between the interval of 511.0 to 512.0, for IEEE-754

(a) single precision

(b) and double precison

numbers.

5) Summing a long list of numbers

On the lecture's moodle-site, you'll find a binary file *numbers.dat* (8 MB). This contains first a 32-bit integer number that gives the number of double-precision values stored in the file (1 million in the provided example), followed by the numbers themselves. (The file is in little-endian. If you happen to work on a big endian processor, which is unlikely these days, you need to swap the endianness.)

Write a read-statement for these numbers, and then try to sum them up, using different approaches.

(a) First, sum the numbers with a simple loop, sequentially from the beginning to the end. Write down the result.

(b) Next, sum them from the end to the beginning, reversing the initial direction of the loop. Write down the result.

(c) Sort the numbers by their magnitude, and sum them from small to large. What do you get now?

(d) Repeat the last experiment by using a summation variable of type `long double`. Do you think the obtained result is trustworthy and correct? (Optional: Try to get real 'quad-double' precision to work where the machine epsilon is of the order of 10^{-35} or so. Check the manuel of your compiler what `long double` actually stands for... and about how you can switch on real 128-bit floating point precison emulated in software.)

(e) Write a program that *exactly* sums the list of numbers using the GMP library (GNU big number library) that allows arbitrary precision floating point accuracy. What do you get?