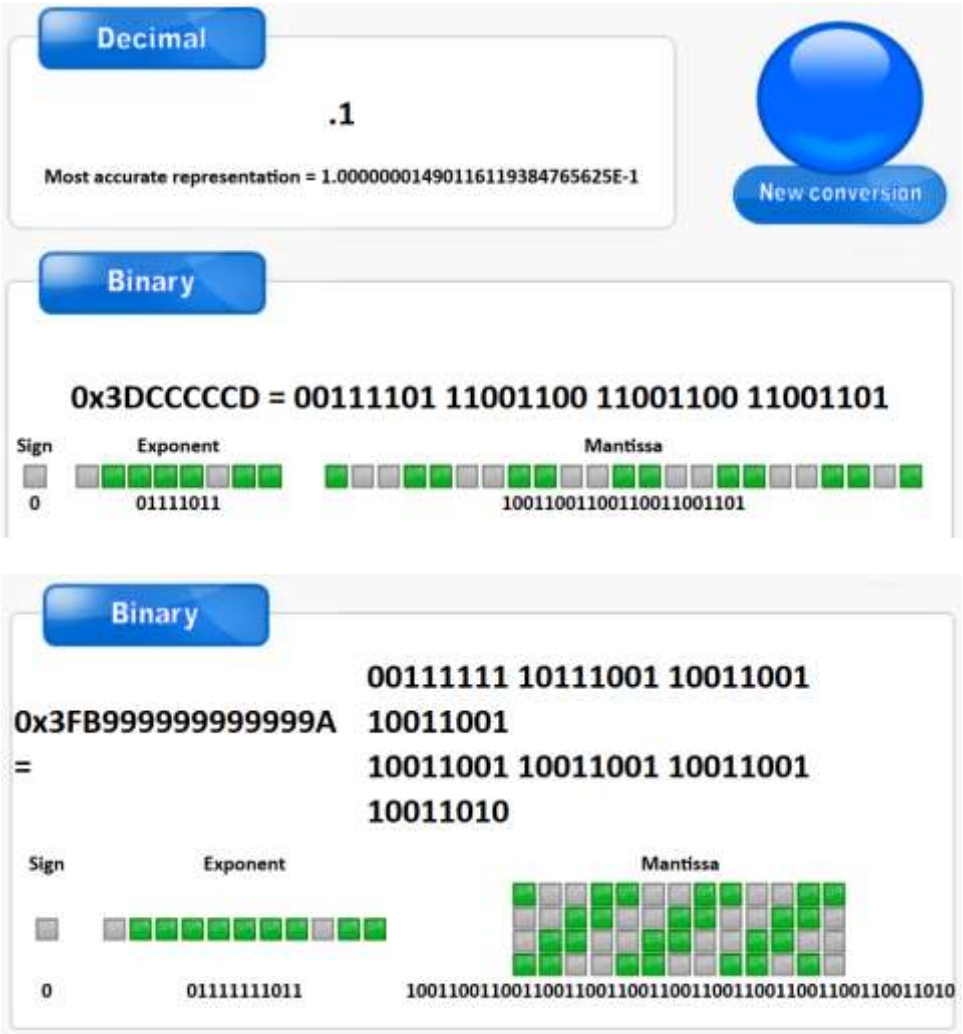
Beware of floating point arithmetics

See additional documents at http://www.isima.fr/~hill/HPC/HPC-Lab3-Docs/

Introduction

There are several traps that even very experienced programmers fall into when they write code that depends on floating point arithmetic. This article explains five things to keep in mind when working with floating point numbers, i.e. `float` and `double` data types.

The study of the IEEE 754 standard for coding real numbers in a limited set of bits is often forgotten by computer scientists. When they start to produce wrong results with their favorite computers they dig again in their past lectures… For a fast memory "recall", see at the following URL how floating points numbers are coded in binary : http://www.binaryconvert.com/ . Like 1/3 which cannot be represented exactly in decimal, 0.1 cannot be represented exactly in binary (the binary numbers ends by the following motif repeated infinitely: 1100 1100 1100 which has to finally be rounded to store a number in 32 or 64 bits. The first picture gives the representation in float with 32 bits and the section in double -64 bits).



How does it impact my computing?

Let's have a look at the code you have studied during the last lecture. Even if you now know that it returns "only 1 is right", you should try to get a deep understanding of how things work.

```
int main()
{
    float a = 0.7;
    float b = 0.5;

    if (a < 0.7)
    {
       if (b < 0.5) printf("2 conditions are right");
       else printf("only 1 is right");
    }
    else printf("0 conditions are right");
}
```

The code is kept simple in C but most languages (often relying on C) will show the same behavior. This behavior is 100% predictable assuming you are using the IEEE 754 standard for floating point. This should be the case for every numerical computing, but you will see that this fine rule has been neglected by many, including in the HPC domain.

Floats get promoted to doubles during comparison, and since floats are less precise than doubles, 0.7 as float is not the same as 0.7 as double.

In this case, 0.7 as float becomes inferior to 0.7 as double when it gets promoted.

And as 0.5 is a power of 2, it is always represented exactly, so the test works as expected: 0.5 < 0.5 is false.

If you want the naive expected behavior, change float to double or .7 and .5 to .7f and .5f and you will get the expected behavior.

Test the following code, to realize how different the representations in float and double are.

```
int main()
{
    float  a = 0.7, b = 0.5; // These are FLOATS
    double c = (double) a;
    double d = 0.7;

    printf("%0.10f", a);
    printf("%0.15f", c);

    printf("%0.15f", d);
    printf("%0.18f", c);
    printf("%0.18", d);

    if (a < .7)                                 // This is a DOUBLE
    {
      if (b < .5) printf("2 are right");        // Here too the test is a DOUBLE
      else printf("1 is right");
    }
    else printf("0 are right");
}
```

Don't Test for Equality

Experienced developers (almost) never want to write code like the following:

```
double x;
double y;
...
if (x == y) {...}
```

Most floating point operations involve at least a tiny loss of precision and so even if two numbers are equal for all practical purposes, they may not be exactly equal down to the last bit, and so the equality test is likely to fail. For example, the following code snippet prints `-1.778636e-015`. Although in theory, squaring should undo a square root, the round-trip operation is slightly inaccurate.

```
double x = 10;
double y = sqrt(x);
y *= y;
if (x == y)
    cout << "Square root is exact\n";
else
    cout << x-y << "\n";
```

In most cases, the equality test above should be written as something like the following:

```
double tolerance = ...
if (fabs(x - y) < tolerance) {...}
```

Here `tolerance` (often noted `eps` or `epsilon`) is some threshold that defines what is "close enough" for equality. This begs the question of how close is close enough. Even if you can quickly consider that this `epsilon` can be set to `1e-6` for single precision and `1e-12` for double precision, this cannot be answered exactly in this short presentation. There are ready to use constants in *float.h* like FLT_EPSIPLON and DBL_EPSIPLON that you can use. However you have to know something about your particular problem to know how close is close enough in your context.

Relative errors

An error of 0.00001 is appropriate for numbers around one, too big for numbers around 0.00001, and too small for numbers around 10,000. A more generic way of comparing two numbers – that works regardless of their range, is to check the relative error. Relative error is measured by comparing the error to the expected result. One way of calculating it would be like this:

```
relativeError = fabs((result - expectedResult) / expectedResult);
```

If result is 99.5, and expectedResult is 100, then the relative error is 0.005.

Worry about Addition and Subtraction more than Multiplication and Division

The relative errors in multiplication and division are always small. Addition and subtraction, on the other hand, can result in complete loss of precision.

This fact leads to a major problem encountered in many HPC applications. The reduction phase is delicate and you may have inaccurate, false or non-reproducible results just because of the final reduction phase after an intensive computing that you may have achieved with the highest numerical quality.

Addition by itself can be a problem (see the differences in the code below), but the more significant problem is subtraction; addition can only be a problem when the two numbers being added have opposite signs, so you can think of that as subtraction. Still, code might be written with a "+" that is really subtraction and it is often said that we have problems with “additions” in the reduction phase of a parallel code.

```
float f     = 0.1f;
float sum   = 0 ;
```

```
for (int i = 0; i < 10; ++i) sum += f;

float product = f * 10;

printf("sum = %1.15f, mul = %1.15f, mul2 = %1.15f\n", sum, product, f * 10);
```

Subtraction is a problem when the two numbers being subtracted are nearly equal. The more nearly equal the numbers, the greater the potential for loss of precision. Specifically, if two numbers agree to n bits, n bits of precision may be lost in the subtraction. This may be easiest to see in the extreme: If two numbers are not equal in theory but they are equal in their machine representation, their difference will be calculated as zero, 100% loss of precision.

Here's an example where such loss of precision comes up often. The derivative of a function `f` at a point `x` is defined to be the limit of `(f(x+h) - f(x))/h` as `h` goes to zero. So a natural approach to computing the derivative of a function would be to evaluate `(f(x+h) - f(x))/h` for some small `h`. In theory, the smaller `h` is, the better this fraction approximates the derivative. In practice, accuracy improves for a while, but past some point smaller values of `h` result in worse approximations to the derivative. As `h` gets smaller, the approximation error gets smaller but the numerical error increases. This is because the subtraction `f(x+h) - f(x)` becomes problematic. If you take `h` small enough (after all, in theory, smaller is better) then `f(x+h)` will equal `f(x)` to machine precision. This means all derivatives will be computed as zero, no matter what the function, if you just take `h` small enough. Here's an example computing the derivative of `sin(x)` at `x = 1`.

```
cout << std::setprecision(15);
for (int i = 1; i < 20; ++i)
{
    double h = pow(10.0, -i);
    cout << (sin(1.0+h) - sin(1.0))/h << "\n";
}
cout << "True result: " << cos(1.0) << "\n";
```

Here is the output of the C++ code above. To make the output easier to understand, digits after the first incorrect digit have been replaced with periods.

```
0.4..........
0.53.........
0.53.........
0.5402.......
0.5402.......
0.540301.....
0.5403022....
0.540302302..
0.54030235...
0.5403022....
0.540301.....
0.54034......
0.53.........
0.544........
0.55.........
0
0
0
0
True result: 0.54030230586814
```

The accuracy improves as `h` gets smaller until $h = 10^{-8}$. Past that point, accuracy decays due to loss of precision in the subtraction. When $h = 10^{-16}$ or smaller, the output is exactly zero because `sin(1.0+h)` equals `sin(1.0)` to machine precision. (In fact, `1+h` equals `1` to machine precision. More on that below.)

(The results above were computed with Visual C++ 2008. When compiled with gcc 4.2.3 on Linux, the results were the same except of the last four numbers. Where VC++ produced zeros, gcc produced negative numbers: -0.017..., -0.17..., -1.7..., and 17....)

What do you do when your problem requires subtraction and it's going to cause a loss of precision? Sometimes the loss of precision isn't a problem; `double`s start out with a lot of precision to spare. When the precision is important, it's often possible to use some trick to change the problem so that it doesn't require subtraction, or doesn't require the same subtraction that you started out with.

See the CodeProject article Avoiding Overflow, Underflow, and Loss of Precision for an example of using algebraic trickery to change the quadratic formula into form more suitable for retaining precision. See also comparing three methods of computing standard deviation for an example of how algebraically equivalent methods can perform very differently.

Floating Point Numbers have Finite Ranges

Everyone knows that floating point numbers have finite ranges, but this limitation can show up in unexpected ways. For example, you may find the output of the following lines of code surprising.

```
float f = 16777216;
cout << f << " " << f+1 << "\n";
```

This code prints the value `16777216` twice. What happened? According to the IEEE specification for floating point arithmetic, a `float` type is 32 bits wide. Twenty four of these bits are devoted to the significand (what used to be called the **mantissa**) and the rest to the exponent. The number `16777216` is 2^{24} and so the `float` variable `f` has no precision left to represent `f+1`. A similar phenomena would happen for 2^{53} if `f` were of type `double` because a 64-bit `double` devotes 53 bits to the significand. The following code prints `0` rather than `1`.

```
x = 9007199254740992; // 2^53
cout << ((x+1) - x) << "\n";
```

We can also run out of precision when adding small numbers to moderate-sized numbers. For example, the following code prints "`Sorry!`" because `DBL_EPSILON` (defined in *float.h*) is the smallest positive number e such that 1 + e != 1 when using `double` types.

```
x = 1.0;
y = x + 0.5 * DBL_EPSILON;
if (x == y)
    cout << "Sorry!\n";
```

Similarly, the constant `FLT_EPSILON` is the smallest positive number e such that 1 + e is not 1 when using `float` types.

Use Logarithms to Avoid Overflow and Underflow

The limitations of floating point numbers described in the previous section stem from having a limited number of bits in the significand. Overflow and underflow result from also having a finite number of bits in the exponent. Some numbers are just too large or too small to store in a floating point number.

Many problems appear to require computing a moderate-sized number as the ratio of two enormous numbers. The final result may be representable as a floating point number even though the intermediate results are not. In this case, logarithms provide a way out. If you want to compute `M/N` for large numbers `M`

and N, compute log(M) - log(N) and apply exp() to the result. For example, probabilities often involve ratios of factorials, and factorials become astronomically large quickly. For N > 170, N! is larger than DBL_MAX, the largest number that can be represented by a double (without extended precision). But it is possible to evaluate expressions such as 200!/(190! 10!) without overflow as follows:

```
x = exp( logFactorial(200)
       - logFactorial(190)
       - logFactorial(10) );
```

A simple but inefficient logFactorial function could be written as follows:

```
double logFactorial(int n)
{
    double sum = 0.0;
    for (int i = 2; i <= n; ++i)
        sum += log((double)i);
    return sum;
}
```

A better approach would be to use a log gamma function if one is available. See How to calculate binomial probabilities for more information.

Numeric Operations don't Always Return Numbers

Because floating point numbers have their limitations, sometimes floating point operations return "infinity" as a way of saying "the result is bigger than I can handle." For example, the following code prints 1.#INF on Windows and inf on Linux.

```
x = DBL_MAX;
cout << 2*x << "\n";
```

Sometimes the barrier to returning a meaningful result has to do with logic rather than finite precision. Floating point data types represent real numbers (as opposed to complex numbers) and there is no real number whose square is -1. That means there is no meaningful number to return if code requests sqrt(-2), even in infinite precision. In this case, floating point operations return NaNs (which stands for “Not a Number). These are floating point values that represent error codes rather than numbers. NaN values display as 1.#IND on Windows and nan on Linux.

Once a chain of operations encounters a NaN, everything is a NaN from there on out. For example, suppose you have some code amounts leading to something like the following:

```
if (x - x == 0) // do something
```

What could possibly keep the code following the if statement from executing? If x is a NaN, then so is x - x and NaNs don't equal anything. In fact, NaNs don't even equal themselves. That means that the expression x == x can be used to test whether x is a (possibly infinite) number. For more information on infinities and NaNs, see IEEE floating point exceptions in C++.

For More Information

This article has be composed with an example found on “Stack overflow” and mostly with the following article with additional comments for the Master MSIR course.

http://www.codeproject.com/Articles/29637/Five-Tips-for-Floating-Point-Programming

The article What Every Computer Scientist Should Know About Floating-Point Arithmetic explains floating point arithmetic in great detail. It may be what every computer scientist would know ideally, but very few will absorb everything presented there.

Remember: Optimisations inside micro porcessor using out order instructions (that do not respect the order specified by the programmer) can damage numerical reproducibility and also the final result.

Floating point arithmetic does not operate in R but in Q (set of fractions) and thus is not associative for addition and multiplication:

a + (b + c) does not equal (a + b) + c !!

Example :

$$(10^{-3} + 1) - 1 \sim 0$$
$$10^{-3} + (1 - 1) = 10^{-3}$$

```
>>> (pow(10,-3)+1)-1
0.0009999999999998899
>>> pow(10,-3)+(1-1)
0.001
>>>
```

Another example: The expression (2^{60} - 2^{60}) + 1 equals 0 + 1, which ultimately equals 1. In this case, the result is correct because the subtraction is done as expected, but if the order of the operations is changed as in (2^{60} + 1) - 2^{60}, the result will be false and will give 0.

With the double precision representation, only 53 bits are available for the mantissa and the expression 2^{60} + 1 will be rounded to 2^{60}. So that after adding one, we still get 2^{60} and removing 2^{60}, will give 0, a false result.

The loss is smother for multiplication except when dealing with overflows).