TP 04 : Integers and floating point

Introduction: This lab is devoted to exercises on integers and floating-point representation numbers.

1 A bit of steganography

Steganography is the *art* of hiding text in plain view. There are many ways to do it, but the one we are interested in is quite simple and related to the topic of TP. Your images can be encoded in several ways (hence the different format : `jpeg`, `png`, `gif`, ...). But the idea remains the same : an image is a rectangle composed of pixels. Each pixel is itself divided into three components : **R**ed, **G**reen, **B**lue. These components generally take values between 0 and 255, so we can encode a total of 2^{24} colors ! However, to the naked eye, it is impossible to distinguish between all these colors. The idea is to alter very slightly the pixels of the image to store our message. For that, we will consider that our message is a a string containing 0 and 1. Then, we will go through our pixels one by one and for each component, the lsb will take as value the next bit of the message to display.

For example, suppose we have a pixel whose RGB components are : $19, 38, 209$, which corresponds to this color. If the first three bits that we wish to encode are 011 then the new pixel will be : $18, 39, 209$, which corresponds to this color, do you see a difference ?

Exercise :

- In the following address : `http://www.lsv.fr/~khmelnitsky/teaching/2019-2020/arch_sys/tp5-files/hidden_text.png` , you will find an image in which a message has been hidden with the technique presented above. Can you find the content of the message ?
- Create a function that allows to do the opposite operation, namely given an image and a text, hides this text in the image.

You are free to use any programming language of you want. But I recommend you to use python(since I'd be able to help you with the language if you get stuck), and to use the following skeleton code : `www.lsv.fr/~khmelnitsky/teaching/2019-2020/arch_sys/tp5-files/image_skeleton.py`

2 Binary operations

The C language has bit manipulation mechanisms. For example, consider two integer variables x and y and the operator $\oplus$ (xor). Let x_i and y_i be the i^{th} bit of x and y respectively. The result of $x \oplus y$ is the word z such that $z_i = x_i \oplus y_i$. The operators in C are `&` (and), `|` (or), `^` (xor) and `~` (not).

Note : do not confuse logical operators such as `&&`, `||`, etc., with the binary operators. Indeed, whereas `4&2` is 0, `4&&2` is 1.

Binary operations can be condensed. So `x=x|2` can be written `x|=2` , and `x=x^y` can be written `x^=y`. The language also provides shift operators to the right >> or left <<.

1. What does the following code do ? :

```
n & (n-1)
```

2. In the following code c and n are integers :

   ```
   for (c = 0; n != 0; n &= (n-1)) c++;
   ```

   What is the value of c after the for loop ?

3. We now construct a method that effectively counts the number of 1 in a word of length 2^k (for a $k \geq 0$), i.e. in $\mathcal{O}(k)$ operations, assuming that 2^k is the size of a register.
   Let $l \leq k$ and n be a word of length 2^k. We call l-*block* a partition of n to consecutive bits of length 2^l, such that the blocks do not overlap. (For example, there are eight 2-blocks of length 4 in a 32-bit word.) The l-*count* of n is the word of length 2^k such that each its of its l-blocks contains the number of 1 of the same l-block in n. For example, the 1-count of $00\ 01\ 10\ 11$ is $00\ 01\ 01\ 10$. Trivially, every word equals its own 0-count. In the following we generate the k-count of n.
   We will assume that $k = 5$ and work with 32-bit registers. Although the method is easy to generalize for greater values of k.

   (a) Find an operation that produces the 1-count of n (in constant time).

   (b) Generalize and iterate this operation to calculate the 5-count of n.

4. In the following we work with 64-bit registers. Let $n = (stuvwxyz)_2$ be one byte(word of length 8), where s is the most significant bit and z the least significant. What does the following expression in C give us ?

   ```
   (n * 0x0202020202 & 0x010884422010) % 1023
   ```

   You may use the following code for experimentation :
   `www.lsv.fr/~khmelnitsky/teaching/2019-2020/arch_sys/tp5-files/bits.c`

3 Quake III Arena bit-hack

In the source code of Quake 3 you can find the following code (the name of the function was redacted by yours truly) :

```
float *******( float number )
{
1     long i;
2     float x2, y;
3     const float threehalfs = 1.5F;
4
5     x2 = number * 0.5F;
6     y  = number;
7     i  = * ( long * ) &y;                       // evil floating point bit level hacking
8     i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
9     y  = * ( float * ) &i;
10    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
11    //y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed
12
13    return y;
}
```

The comments were left by a fellow programmer who as us was completely baffled by this code. Let's try to understand what is happening here, but first :

Newton's method :
Assume we want to find an approximation to $\sqrt(2)$, we could reformulate this problem to finding the root of

the function $f(x) = x^2 - 2$ (i.e. $f(x) = 0$). Newton's method give us a way to recursively approximate the solution to this kind of problems (where we assume that f is 'nice enough' near the wanted root). The idea is to start with an initial guess which is reasonably close to the true root, then to approximate the function by its tangent line using calculus, and finally to compute the intersection with the x axis of this tangent line by elementary algebra. This will typically be a better approximation to the original function's root than the first guess, and the method can be iterated.

More formally, given some guess for an x_n which is "close" to the root, we take the tangent line to our curve $y = f(x)$ at x_n and find it's intersection with the x axis : (maybe add a drawing)

$$0 = f'(x_n)(x - x_n) + f(x_n)$$

By moving the variables around we get a better approximation for the root :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

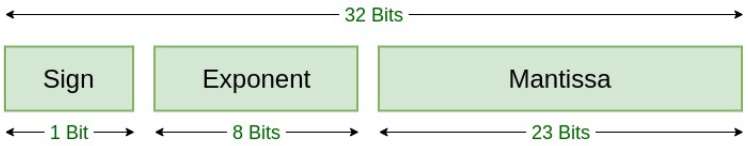
Exercise :

1. What is x_{n+1} for the following functions : a. $f(x) = x^2 - a$; b. $f(x) = 1/x^2 - a$
2. Explain lines 10-11 of the code, and conclude the purpose of the function.
3. Can you guess what such a function would be used for in a game like Quake(a 3D shooter) ?

Finding a good initial solution :

A good initial guess would help the Newton's method to work fast. Note that in the code they are so sure of their initial guess that they run only one iteration of Newton's method.

Recall that in C, the type `float` represents real values according to the IEEE 754 standard in the 32 bit variant, with 1 bit for the sign(which we'll denote by S it's integer value), 8 for the exponent(which we will denote by E) and 23 for the mantissa(which we will denote by M its integer value).



Single Precision
IEEE 754 Floating-Point Standard

Denote by $L = 2^{23}$, $B = 127$ and :

$$m = \frac{M}{L} \;\; ; \;\; e = E - B$$

Since we are working with square roots, we assume that we are working with non negative numbers, i.e. $S = 0$.

Exercise :

1. Show that $(1 + m)2^e$ is the number corresponding to the value of the floating point.
2. Given an encoding of a floating point with mantissa M and exponent E, show that if we had decoded it as an integer its value would be $M + LE$.

We want to find an approximation to $y = x^{-1/2}$, or in our new terms :

$$(1 + m_y)2^{e_y} = ((1 + m_x)2^{e_x})^{-1/2}$$

We can take log from both sides, to start disentangling the equation above :

$$\log(1 + m_y) + e_y = -\frac{1}{2}(\log(1 + m_x) + e_x) \tag{1}$$

But now we need to somehow take care of the $\log$. For this we take another approximation $\log(1+v) \approx v+\sigma$, where $0 \leq v \leq 1$ and σ is a constant that needs to be chosen carefully.

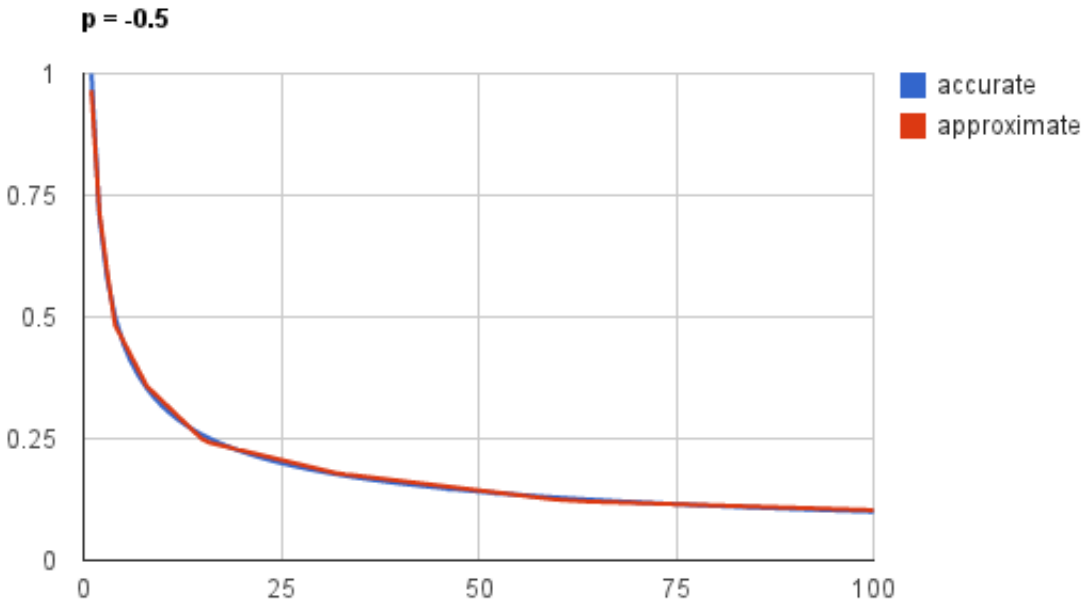
Exercise :

1. Combine equation 1, the approximation of the $\log$ function, and the original values of the mantissa and exponent to get :
$$M_y + LE_y \approx \frac{3}{2}L(B - \sigma) - \frac{1}{2}(M_x + LE_x)$$

2. Let $\sigma = 0.0450465$, explain lines 5-8 in the code

Unfortunately, the explanation for the specific value of σ is probably trial and error.

Even though this method may seem very strange and inaccurate, it turns out to be a very fast and a very good approximation in practice :



Exercise : This approximation can be done for any $f(x) = x^c + a$ where $-1 < c < 1$. Rewrite the code for $c = 0.5$.

4 Floating point

For this exercise you can find a skeleton code here :
`www.lsv.fr/~khmelnitsky/teaching/2019-2020/arch_sys/tp5-files/float-skel.c`
We consider the following type which is represented by three integers :

```
typedef struct { int sign; int exponent; int mantissa; } fc;
```

1. Write a C function that splits a `float` into its three components of `fc`. For example, the IEEE 754 representation of 2.5 is :

```
0 . 1000 0000 . 010 0000 0000 0000 0000 0000
```

In this case, the returned structure would contain `sign` $= 0$, `exposant` $= 128$ and `mantisse` $= 2097152$.

Reminder : To do this, it is useful to use the conversion of types in C (*typecast*), e.g. `(int)f`, where `f` is a `float`, interprets the binary contents of `f` as an integer. Make sure that `int` and `float` are of the same size on your machine !

2. Create a function that does the opposite, ie returns the `float` corresponding to a given `fc`.
3. Implement addition on `fc`. For simplicity, we will make the following restrictions : (i) the two operands are positive ; (ii) special cases NaN/Inf etc. are not treated.

   The addition for `fc` is done in three steps :

   (a) Standardizing the two values, i.e. if the two exponents are different, we adjust the mantissa of one of the two according to the difference.

   (b) Sum the two mantissas, taking into account the “hidden” bit representing the 1.

   (c) Normalize the mantissa to put it in the interval $[1, 2)$ and adjust the exponent of the result accordingly.