C++ Implementation

Pieter P

Dividing by Powers of 2

The factor α in the difference equation of the Exponential Moving Average filter is a number between zero and one. There are two main ways to implement this multiplication by α: Either we use floating point numbers and calculate the multiplication directly, or we use integers, and express the multiplication as a division by $1/\alpha > 1$.
Both floating point multiplication and integer division are relatively expensive operations, especially on embedded devices or microcontrollers.

We can, however, choose the value for α in such a way that $1/\alpha = 2^k, k \in \mathbb{N}$.
This is useful, because a division by a power of two can be replaced by a very fast right bitshift:

$$\alpha \cdot x = \frac{x}{2^k} = x \gg k$$

We can now rewrite the difference equation of the EMA with this optimization in mind:

$$\begin{aligned} y[n] &= \alpha x[n] + (1-\alpha)y[n-1] \\ &= y[n-1] + \alpha\left(x[n] - y[n-1]\right) \\ &= y[n-1] + \frac{x[n] - y[n-1]}{2^k} \\ &= y[n-1] + (x[n] - y[n-1]) \gg k \end{aligned}$$

Negative Numbers

There's one caveat though: this doesn't work for negative numbers. For example, if we try to calculate the integer division $-15/4$ using this method, we get the following answer:

$$\begin{aligned} -15/4 &= -15 \cdot 2^{-2} \\ -15 \gg 2 &= 0b11110001 \gg 2 \\ &= 0b11111100 \\ &= -4 \end{aligned}$$

This is not what we expected! Integer division in programming languages such as C++ returns the quotient truncated towards zero, so we would expect a value of -3. The result is close, but incorrect nonetheless.

This means we'll have to be careful not to use this trick on any negative numbers. In our difference equation, both the input $x[n]$ and the output $y[n]$ will generally be positive numbers, so no problem there, but their difference can be negative. This is a problem. We'll have to come up with a different representation of the difference equation that doesn't require us to divide any negative numbers:

$$\begin{aligned} y[n] &= y[n-1] + \alpha\left(x[n] - y[n-1]\right) \\ y[n] &= y[n-1] + \frac{x[n] - y[n-1]}{2^k} \\ 2^k y[n] &= 2^k y[n-1] + x[n] - y[n-1] \end{aligned}$$

$$z[n] \triangleq 2^k y[n] \Leftrightarrow y[n] = 2^{-k} z[n]$$

$$z[n] = z[n-1] + x[n] - 2^{-k} z[n-1]$$

We now have to prove that $z[n-1]$ is greater than or equal to zero. We'll prove this using induction:

Base case: $n - 1 = -1$

The value of $z[-1]$ is the initial state of the system. We can just choose any value, so we'll pick a value that's greater than or equal to zero: $z[-1] \geq 0$.

Induction step: n

Given that $z[n-1] \geq 0$, we can now use the difference equation to prove that $z[n]$ is also greater than zero:

$$z[n] = z[n-1] + x[n] - 2^{-k} z[n-1]$$

We know that the input $x[n]$ is always zero or positive.
Since $k > 1 \Rightarrow 2^{-k} < 1$, and since $z[n-1]$ is zero or positive as well, we know that
$z[n-1] \geq 2^{-k} z[n-1] \Rightarrow z[n-1] - 2^{-k} z[n-1] \geq 0$.
Therefore, the entire right-hand side is always positive or zero, because it is a sum of two numbers that are themselves greater than or equal to zero. $\square$

Rounding

A final improvement we can make to our division algorithm is to round the result to the nearest integer, instead of truncating it towards zero.
Consider the rounded result of the division a/b. We can then express it as a flooring of the result plus one half:

$$\begin{aligned}\left\lfloor \frac{a}{b} \right\rceil &= \left\lfloor \frac{a}{b} + \frac{1}{2} \right\rfloor \\ &= \left\lfloor \frac{a + \frac{b}{2}}{b} \right\rfloor\end{aligned}$$

When b is a power of two, this is equivalent to:

$$\begin{aligned}\left\lfloor \frac{a}{2^k} \right\rceil &= \left\lfloor \frac{a}{2^k} + \frac{1}{2} \right\rfloor \\ &= \left\lfloor \frac{a + \frac{2^k}{2}}{2^k} \right\rfloor \\ &= \left\lfloor \frac{a + 2^{k-1}}{2^k} \right\rfloor \\ &= (a + 1 \ll (k-1)) \gg k\end{aligned}$$

Implementation in C++

We now have everything in place to write an implementation of the EMA in C++:

```cpp
#include <stdint.h>

template <uint8_t K, class uint_t>
class EMA {
  public:
    uint_t filter(uint_t x) {
        z += x;
        uint_t y = (z + (1 << (K - 1))) >> K;
        z -= y;
        return y;
    }

  private:
    uint_t z = 0;
};
```

Note how we save $z[n] - 2^{-k}z[n]$ instead of just $z[n]$. Otherwise, we would have to calculate $2^{-k}z[n]$ twice (once to calculate $y[n]$, and once on the next iteration to calculate $2^{-k}z[n-1]$), and that would be unnecessary.

Signed Rounding Division

It's possible to implement a signed division using bit shifts as well. The only difference is that we have to subtract 1 from the dividend if it's negative.

On ARM and x86 platforms, the performance difference between the signed and unsigned version is small, because determining whether the dividend is negative can easily be done by shifting the sign bit to the least significant place (a right shift of 31 bits).

On architectures where bit shifts are more expensive, like the AVR architecture used by Arduino microcontrollers, the signed version is much slower.
The reason for this is that the AVR architecture only has an instruction to shift a single bit, not to shift an arbitrary number of bits. Determining the sign of a number is therefore more complicated.

Implementation of Signed and Unsigned Division by a Multiple of Two

```cpp
constexpr unsigned int K = 3;

signed int div_s(signed int val) {
    int neg = val < 0 ? 1 : 0;
    int shiftval = (val + (1 << (K - 1)) - neg) >> K;
    return shiftval;
}

unsigned int div_u(unsigned int val) {
    return (val + (1 << (K - 1))) >> K;
}
```

Assembly Generated on x86_64

```
div_s(int):
        lea     eax, [rdi+4]
        shr     edi, 31
        sub     eax, edi
        sar     eax, 3
        ret
div_u(unsigned int):
        lea     eax, [rdi+4]
        shr     eax, 3
        ret
```

Assembly Generated on ARM

```
div_s(int):
        add     r3, r0, #4
        sub     r0, r3, r0, lsr #31
        asr     r0, r0, #3
        bx      lr
div_u(unsigned int):
        add     r0, r0, #4
        lsr     r0, r0, #3
        bx      lr
```

Assembly Generated on AVR

```
__zero_reg__ = 1
div_s(int):
        mov r18,r24
        mov r19,r25
        subi r18,-4
        sbci r19,-1
        mov r24,r25
        rol r24
        clr r24
        rol r24
        mov r20,r18
        mov r21,r19
        sub r20,r24
        sbc r21,__zero_reg__
        mov r24,r20
        mov r25,r21
        asr r25
        ror r24
        asr r25
        ror r24
        asr r25
        ror r24
        ret
div_u(unsigned int):
        adiw r24,4
        lsr r25
        ror r24
        lsr r25
        ror r24
        lsr r25
        ror r24
        ret
```

Keep in mind that an `int` on AVR is only 16 bits wide, whereas an `int` on ARM or x86 is 32 bits wide.
If you use 32-bit integers on AVR, the result is even more atrocious.

You can try it for yourself on the Compiler Explorer.
