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:

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

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:

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

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

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

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 a basic implementation of the EMA in C++:

```cpp
#pragma once
#include <cstdint>     // uint8_t, uint16_t
#include <type_traits> // std::is_unsigned

/// The first Exponential Moving Average implementation for unsigned integers.
///
/// @note   An improved implementation is presented further down the page.
template <uint8_t K, class uint_t = uint16_t>
class EMA {
  public:
    /// Update the filter with the given input and return the filtered output.
    uint_t operator()(uint_t input) {
        state += input;
        uint_t output = (state + half) >> K;
        state -= output;
        return output;
    }

    static_assert(std::is_unsigned<uint_t>::value,
                  "The `uint_t` type should be an unsigned integer, "
                  "otherwise, the division using bit shifts is invalid.");

    /// Fixed point representation of one half, used for rounding.
    constexpr static uint_t half = 1 << (K - 1);

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

Note how we save $z[n] - 2^{-k}z[n]$ as the state, 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 absolute performance difference between the signed and unsigned version is not too big, it requires just a few more instructions. However, on some other architectures, like the AVR architecture used by some Arduino microcontrollers, the division is by far the most expensive step of the EMA algorithm, so a slower signed division might have a significant impact on the overall performance. In theory, it should only take a couple instructions to conditionally subtract 1, based on the sign of the dividend, but this sometimes causes the compiler to refactor the entire division, resulting in a much slower algorithm.

I provided two implementations of the signed division. Notice how on x86 and ARM the second one is faster, while on AVR, the first one is faster. The unsigned division is included for reference.

The code was compiled using the `-O2` optimization level.

Implementation of Signed and Unsigned Division by a Multiple of Two

```
constexpr unsigned int K = 3;

signed int div_s1(signed int val) {
    int round = val + (1 << (K - 1));
    if (val < 0)
        round -= 1;
    return round >> K;
}

signed int div_s2(signed int val) {
    int neg = val < 0 ? 1 : 0;
    return (val + (1 << (K - 1)) - neg) >> K;
}

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

Assembly Generated on x86_64 (GCC 9.2)

```
div_s1(int):
        mov     eax, edi
        not     eax
        shr     eax, 31
        lea     eax, [rax+3+rdi]
        sar     eax, 3
        ret

div_s2(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 64 (GCC 8.2)

```
div_s1(int):
        mvn     w1, w0
        add     w0, w0, w1, lsr 31
        add     w0, w0, 3
        asr     w0, w0, 3
        ret

div_s2(int):
        add     w1, w0, 4
        sub     w0, w1, w0, lsr 31
        asr     w0, w0, 3
        ret

div_u(unsigned int):
        add     w0, w0, 4
        lsr     w0, w0, 3
        ret
```

Assembly Generated on AVR (GCC 5.3)

```
div_s1(int):
        sbrc r25,7  # Skip if Bit in Register is Cleared: val >= 0
        rjmp .L2
                    # val >= 0
        adiw r24,4  # Add Immediate to Word: val + (1 << (K - 1)) = val + 4
        asr r25     # Arithmetic Shift Right: shift high byte (preserve sign)
        ror r24     # Rotate Right through Carry: shift low byte
        asr r25     # Two more times
        ror r24
        asr r25
        ror r24
        ret
.L2:
                    # val < 0
        adiw r24,3  # Add Immediate to Word: val + (1 << (K - 1)) - 1 = val + 3
        asr r25     # Arithmetic Shift Right: shift high byte (preserve sign)
        ror r24     # Rotate Right through Carry: shift low byte
        asr r25     # Two more times
        ror r24
        asr r25
        ror r24
        ret

div_s2(int):
        movw r18,r24
        subi r18,-4   # Subtract immediate: val + (1 << (K - 1)) = val + 4
        sbci r19,-1   # Subtract Immediate with Carry: (low byte)
        mov r24,r25
        rol r24       # Rotate Left through Carry: C flag is now sign bit
        clr r24       # Clear Register
        rol r24       # Rotate Left through Carry: original sign bit is now lsb
        movw r20,r18
        sub r20,r24   # Subtract without Carry: val + 4 - neg
        sbc r21,__zero_reg__  # Subtract with Carry: (low byte)
        movw r24,r20
        asr r25       # Arithmetic Shift Right: shift high byte (preserve sign)
        ror r24       # Rotate Right through Carry: shift low byte
        asr r25       # Two more times
        ror r24
        asr r25
        ror r24
        ret

div_u(unsigned int):
        adiw r24,4  # Add Immediate to Word: val + (1 << (K - 1)) = val + 4
        lsr r25     # Logical Shift Right: shift high byte (no sign extension)
        ror r24     # Rotate Right through Carry: shift low byte
        lsr r25     # Two more times
        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 experiment with the different implementations yourself on the Compiler Explorer.

The main takeaway from this section is that signed (rounding) division is more expensive than unsigned division.

A better alternative for signed division

Since the EMA is a linear filter, adding a constant offset to the input results in the same output, but with the same offset added to it. This means that we don't have to worry about negative numbers, we can just add a constant offset to the negative inputs, resulting in only positive numbers. At the output of the filter, the offset is simply removed again.

This approach turns out to be significantly more efficient than the signed divisions discussed above. It allows us to use only unsigned rounding divisions, which are very cheap, and just a single extra subtraction to handle signed types. (Yes, just one, it turns out that adding the offset to the input and subtracting it again from the output can be combined.)

The assumption that the EMA is a linear filter is not really valid anymore, because of the rounding and truncation errors introduced by the use of integers in the algorithm. Luckily, the output of the filter turns out to be exactly the same, it doesn't matter if you use true signed rounding division or an unsigned rounding division with offset.

Improved C++ implementation

The following snippet is an improved version of the previous implementation: it supports both signed and unsigned inputs, allows initialization to a specific value, and has a check to prevent overflow.

```cpp
#pragma once
#include <type_traits> // std::make_unsigned_t, make_signed_t, is_unsigned
#include <limits>      // std::numeric_limits
#include <cstdint>     // uint_fast16_t

/**
 * @brief   Exponential moving average filter.
 *
 * Fast integer EMA implementation where the weight factor is a power of two.
 *
 * Difference equation: @f$ y[n] = \alpha·x[n]+(1-\alpha)·y[n-1] @f$
 * where @f$ \alpha = \left(\frac{1}{2}\right)^{K} @f$, @f$ x @f$ is the
 * input sequence, and @f$ y @f$ is the output sequence.
 *
 * [An in-depth explanation of the EMA filter](https://tttapa.github.io/Pages/Mathematics/Systems-and-Control-Theory/Digital-filters/Exponential%20Moving%20Average/)
 *
 * @tparam  K
 *          The amount of bits to shift by. This determines the location
 *          of the pole in the EMA transfer function, and therefore the
 *          cut-off frequency.
 *          The higher this number, the more filtering takes place.
 *          The pole location is @f$ 1 - 2^{-K} @f$.
 * @tparam  input_t
 *          The integer type to use for the input and output of the filter.
 *          Can be signed or unsigned.
 * @tparam  state_t
 *          The unsigned integer type to use for the internal state of the
 *          filter. A fixed-point representation with @f$ K @f$ fractional
 *          bits is used, so this type should be at least @f$ M + K @f$ bits
 *          wide, where @f$ M @f$ is the maximum number of bits of the input.
 *
 * Some examples of different combinations of template parameters:
 *
 * 1. Filtering the result of `analogRead`: analogRead returns an integer
 *    between 0 and 1023, which can be represented using 10 bits, so
 *    @f$ M = 10 @f$. If `input_t` and `output_t` are both `uint16_t`,
 *    the maximum shift factor `K` is @f$ 16 - M = 6 @f$. If `state_t`
 *    is increased to `uint32_t`, the maximum shift factor `K` is
 *    @f$ 32 - M = 22 @f$.
 * 2. Filtering a signed integer between -32768 and 32767: this can be
 *    represented using a 16-bit signed integer, so `input_t` is `int16_t`,
 *    and @f$ M = 16 @f$. (2¹⁵ = 32768)
 *    Let's say the shift factor `K` is 1, then the minimum width of
 *    `state_t` should be @f$ M + K = 17 @f$ bits, so `uint32_t` would be
 *    a sensible choice.
 */
template <uint8_t K,
          class input_t = uint_fast16_t,
          class state_t = std::make_unsigned_t<input_t>>
class EMA {
  public:
    /// Constructor: initialize filter to zero or optional given value.
    EMA(input_t initial = input_t(0))
      : state(zero + (state_t(initial) << K) - initial) {}

    /// Update the filter with the given input and return the filtered output.
    input_t operator()(input_t input) {
      state         += state_t(input);
      state_t output = (state + half) >> K;
      output        -= zero >> K;
      state         -= output;
      return input_t(output);
    }

    constexpr static state_t
      max_state  = std::numeric_limits<state_t>::max(),
      half_state = max_state / 2 + 1,
      zero       = std::is_unsigned<input_t>::value ? state_t(0) : half_state,
      half       = K > 0 ? state_t(1) << (K - 1) : state_t(0);

    static_assert(std::is_unsigned<state_t>::value,
                  "state type should be unsigned");

    static_assert(max_state >= std::numeric_limits<input_t>::max(),
                  "state type cannot be narrower than input type");

    /// Verify the input range to make sure it's compatible with the shift
    /// factor and the width of the state type.
    template <class T>
    constexpr static bool supports_range(T min, T max) {
      using sstate_t = std::make_signed_t<state_t>;
      return min <= max &&
             min >= std::numeric_limits<input_t>::min() &&
             max <= std::numeric_limits<input_t>::max() &&
             (std::is_unsigned<input_t>::value
               ? state_t(max) <= (max_state >> K)
               : min >= -sstate_t(max_state >> (K + 1)) - 1 &&
                 max <= sstate_t(max_state >> (K + 1)));
    }

  private:
    state_t state;
};
```

When the type is signed, an offset of 2^{B-1} is added, where B is the number of bits used to represent the state variable. This essentially shifts the value "zero" up to the middle of the range of the state.

To check the range of the input for specific template parameters, you can use the `supports_range` method:

```cpp
EMA<5, int_fast16_t, uint_fast16_t> filter;
static_assert(filter.supports_range(-1024, 1023),
              "use a wider state or input type, or a smaller shift factor");
```

Arduino Example

```cpp
template <uint8_t K, class uint_t = uint16_t>
class EMA {
  public:
    /// Update the filter with the given input and return the filtered output.
    uint_t operator()(uint_t input) {
        state += input;
        uint_t output = (state + half) >> K;
        state -= output;
        return output;
    }

    static_assert(
        uint_t(0) < uint_t(-1),  // Check that `uint_t` is an unsigned type
        "The `uint_t` type should be an unsigned integer, otherwise, "
        "the division using bit shifts is invalid.");

    /// Fixed point representation of one half, used for rounding.
    constexpr static uint_t half = 1 << (K - 1);

  private:
    uint_t state = 0;
};

void setup() {
  Serial.begin(115200);
  while (!Serial);
}

const unsigned long interval = 10000; // 10000 µs = 100 Hz

void loop() {
  static EMA<2> filter;
  static unsigned long prevMicros = micros() - interval;
  if (micros() - prevMicros >= interval) {
    int rawValue = analogRead(A0);
    int filteredValue = filter(rawValue);
    Serial.print(rawValue);
    Serial.print('\t');
    Serial.println(filteredValue);
    prevMicros += interval;
  }
}
```

Additional resources

The idea to use a constant offset to deal with negative inputs originated in this PJRC forum thread. It also includes a discussion about how the filter works, simulations comparing integer EMA implementations with and without rounding, a pure C implementation, etc.