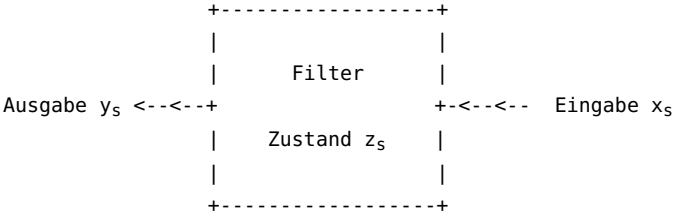
Decay-Filter

Geschrieben von Wolf am 29. Februar 2024.

1. Definition

Ein Filter berechnet einen Ausgabewert y_s aus einem aktuellen Eingabewert x_s und einem internen Zustand z_s, der von den Eingabewerten der Vergangenheit abhängt. Mathematisch ist dies eine Faltung, allerdings rechnen wir nicht kontinuierlich, sondern mit diskreten Schritten $s \in \{0, 1, ...\}$.



Beim Decay-Filter beeinflusst ein Eingabewert die gesamte Zukunft, aber umso schwächer, je weiter ein Zeitpunkt entfernt ist. Der Filter wird durch einen Paramter k definiert mit $0 \leq k < 1$: bei einem großen k bleibt die Wirkung eines Eingabewertes lange erhalten, bei einem kleinen k klingt sie schneller ab. Es gibt eine weitere Konstante k'; diese ergibt sich aber, wie wir noch sehen werden, eindeutig aus k.

Die Ausgabe unseres Filter ist:

$$y_s := k' \cdot \left(1 \cdot x_s + k x_{s-1} + k^2 x_{s-2} + k^3 x_{s-3} + ...\right)$$
$$y_s := k' \cdot \sum_{i=0}^{\infty} k^i \cdot x_{s-i}$$

(1.1) $$y_s := \sum_{i=0}^{\infty} k^i \cdot k' \cdot x_{s-i}$$

2. Mathematik

Eine Forderung an unseren Filter ist, dass bei konstanter Eingabe $x_s := x$ nach hinreichender Zeit die Ausgabe gleich der Eingabe ist:

$$x = k' \cdot \left(1 \cdot x + kx + k^2 x + k^3 x + ...\right)$$
$$x = k' \cdot \sum_{i=0}^{\infty} k^i x$$
$$1 = k' \cdot \sum_{i=0}^{\infty} k^i$$
$$\frac{1}{k'} = \sum_{i=0}^{\infty} k^i$$

Für $|k| < 1$ ergibt sich mit (M.5):

$$\frac{1}{k'} = \frac{1}{1-k}$$

(2.1) $$k' = (1-k)$$

Die Gleichung enthält *alle* Eingangswerte der Vergangenheit x_{s-i}. Wir können sie aber in eine inkrementelle Darstellung umformen, die zur Berechnung von y_s nur x_s und y_{s-1} braucht:

$$y_s = \sum_{i=0}^{\infty} k^i \cdot k' \cdot x_{s-i}$$
$$y_s = k^0 \cdot k' \cdot x_s + \sum_{i=1}^{\infty} k^i \cdot k' \cdot x_{s-i}$$
$$y_s = k^0 \cdot k' \cdot x_s + \sum_{i=0}^{\infty} k^{i+1} \cdot k' \cdot x_{s-(i+1)}$$
$$y_s = k^0 \cdot k' \cdot x_s + k \cdot \underbrace{\sum_{i=0}^{\infty} k^i \cdot k' \cdot x_{(s-1)-i}}_{y_{s-1}}$$

Mit (1.1) und (2.1) erhalten wir die inkrementelle Darstellung:

(2.2) $y_s = (1-k) \cdot x_s + k \cdot y_{s-1} = k \cdot y_{s-1} + (1-k) \cdot x_s$

3. Implementierung mit Ganzzahl-Arithmetik

Die Filtergleichung nutzt die Werte *k* und *1-k*, beide zwischen *0* und *1*. Wir skalieren mit 2^{31} und erhalten den Wertebereich von *0* bis 2^{31}. Durch Verzicht auf die beiden Extremwerte erhalten wir für k_s und *1-*k_s den Bereich *1* bis 2^{31}*-1*, dessen Werte sich als `int32_t` darstellen lassen.

```
static volatile int32_t ks;
```

Wir formen (2.2) um:

$$y_t = k \cdot y_{s-1} + (1-k) \cdot c_t$$

$$y_t = \frac{2^{31}k}{2^{31}} \cdot y_{s-1} + \frac{2^{31} - 2^{31}k}{2^{31}} \cdot c_t$$

$$y_t = \frac{1}{2^{31}} \cdot \left(2^{31}k \cdot y_{s-1} + \left(2^{31} - 2^{31}k\right) \cdot c_t\right)$$

(3.1) $$y_t = \frac{1}{2^{31}} \cdot \left(k_s \cdot y_{s-1} + \left(2^{31} - k_s\right) \cdot c_t\right)$$

Für Eingabe x_s und Ausgabe y_s nutzen wir den Wertebereich von `int32_t`:

```
static int32_t out;
```

Die beiden Multiplikationen haben `int32_t` Argumente, aber ein `int64_t` Ergebnis:

```
static inline int64_t
mul32_64( int32_t a, int32_t b ){

        return (int64_t) a * b;
}
```

Die Summe der beiden Produkte passt in ein `int64_t`.

Die Division duch 2^{31} kann durch ein Shift implementiert werden. Bei negativer Summe muss aber der Absolutbetrag geshifted werden, weil ein Shift negativer Werte von *0* weg rundet und das einen Überlauf verursachen kann.

```
void
decay_filter( int32_t in, int32_t ks, int32_t *outp ){

        // if( ks < 1 ){ ks = 1; }

        const int32_t minus_ks = INT32_MAX - ks + 1;

        const int64_t a = mul32_64( *outp, ks       );
        const int64_t b = mul32_64(  in  , minus_ks );

        const int64_t sum = a + b;

        if( sum >= 0 ){

                const uint64_t abs_sum = (uint64_t) +sum;
                const uint64_t shifted = abs_sum >> 31;

                *outp = +( (int32_t) shifted );
        } else {
                const uint64_t abs_sum = (uint64_t) -sum;
                const uint64_t shifted = abs_sum >> 31;

                *outp = -( (int32_t) shifted );
        }
}
```

4. Reale Implementierung

```
void
decay_filter( int32_t in, int32_t ks, int32_t *outp ){

        const register int64_t sum =
                mul32_64( *outp,            ks ) +
                mul32_64(  in  , INT32_MIN - ks );

        if( sum >= 0 ){

                *outp = +(int32_t)( (+sum) >> 31 );
        } else {
                *outp = -(int32_t)( (-sum) >> 31 );
        }
}

decay_filter( in, ks, &out );
```

Die Funktion benutzt:

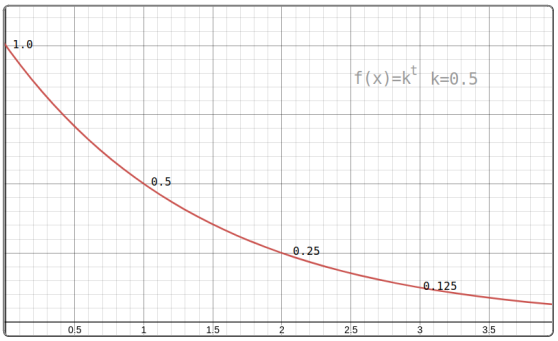
- 1 32-Bit Signed Subtraktion,
- 2 Signed 32-Bit Multiplikationen mit 64-Bit-Ergebnis,
- 1 Signed 64-Bit Addition,
- 1 64-Bit Negation,
- 1 Unsigned 64-Bit Shift, und
- 1 32-Bit Negation.

5. Berechnung von *k* aus der Zeitkonstante

Wir wollen k aus der Halbswertzeit t_h bestimmen. Nach s_h Schritten soll die Wirkung einer Eingabe auf *0.5* abgeklungen sein:

$$k^{s_h} = 0.5$$
$$k = 0.5^{\frac{1}{s_h}}$$
$$k = (1/2)^{\frac{1}{s_h}}$$

(5.1) $$k = 1/2^{\frac{1}{s_h}}$$



Exponentialfunktion mit Basis k=0.5

Wenn die Schritte mit einer konstanten Frequenz f erfolgen, werden in der Zeit t_h die Anzahl $f * t_h$ Schritte durchlaufen:

$$s_h = f \cdot t_h$$

Zusammen mit (5.1) erhalten wir:

(5.2) $$k = 1/2^{\frac{1}{f \cdot t_h}}$$

Zur Vorbereitung der Implementierung wird die Formel in Schritte zerteilt:

(5.3) $$c := \frac{1}{f \cdot t_h}$$

(5.4) $$d := 2^c$$

(5.5) $$k := 1/d$$

- Die Frequenz *f*, mit der die Filterfunktion aufgerufen wird, wird in Hertz (*1/Sekunde*) angegeben.
- Um mit ganzen Zahlen rechnen zu können, geben wir die Halbwertszeit t_h in Mikrosekunden an.

Damit erhalten wir:

(5.3a) $$c := \frac{10^6}{f^{[Hz]} \cdot t_h^{[\mu s]}}$$

6. Implementierung mit Ganzzahl-Arithmetik

Zur Berechnung der Potenz nutzen wir die Funktion *exp2m1(x) := 2^x-1*, eine Verschmelzung aus *exp2* und *expm1*.

Die Ganzzahl-Implementierung *u32p32exp2m1(x)*, wir nennen sie ab hier *p(x)*, hat als Definitionsbereich und Wertebereich *0 ... 1-2^{-32}*, wobei sowohl Argument als auch Ergebnis mit 2^{32} skaliert werden:

(6.1) $p\left(2^{32}x\right) = 2^{32} \cdot (2^x - 1),\ 2^{32}x \in \texttt{uint32\_t} \wedge p\left(2^{32}x\right) \in \texttt{uint32\_t}$

Für die Nutzung dieser Funktion führen wir die skalierten ganzzahligen Variablen c_s, d_s und k_s ein.

Aus (5.3a) wird c_s definiert:

(6.2) $c_s := 2^{32}c = \min\left\{\frac{2^{32} \cdot 10^6}{f^{[Hz]} \cdot t_h^{[\mu s]}}, 2^{32} - 1\right\} \in \texttt{uint32\_t}$

Der verfügbare Wertebereich für c_s erfordert *s_h ≥ 1+1/(2^{32}-1)*, Die Filterroutine muss also etwas häufiger als einmal je Sekunde aufgerufen werden. Diese Bedingung wird trivial erfüllt.

Aus (5.4) und (6.1) wird d_s definiert:

$$d - 1 = 2^c - 1$$
$$2^{32}(d-1) = 2^{32}(2^c - 1) = p\left(2^{32}c\right) = p(c_s)$$

(6.3) $d_s := 2^{32}(d-1) = p(c_s) \in \texttt{uint32\_t}$

Aus (5.5) wird k_s definiert:

$$2^{31}k = \frac{2^{31}}{d} = \frac{2^{63}}{2^{32}d} = \frac{2^{63}}{\underbrace{2^{32}(d-1)}_{d_s} + 2^{32}} = \frac{2^{63}}{d_s + 2^{32}}$$

(6.4) $k_s := 2^{31}k = \min\left\{\frac{2^{63}}{d_s + 2^{32}}, 2^{31} - 1\right\} \in \texttt{int32\_t}$

Der verfügbare Wertebereich für k_s erfordert *d_s ≥ 2*, mit (6.3) also *d ≥ 1+2^{-31}*, mit (5.4) also *2^c ≥ 1+2^{-31}* oder umgeformt *c ≥ log_2(1+2^{-31})*. Wegen (6.2) ist das erfüllt für *c_s = 2^{32}c ≥ 2^{32} log_2(1+2^{-31}) = 2.89*, also *c_s ≥ 3*, mit *c_s = 2^{32} / s_h* demnach für *s_h ≤ 2^{32} / 3 ~ 1.4·10^9*. Die Filterroutine darf also maximal 1.4 Milliarden mal je Sekunde aufgerufen werden. Auch diese Bedingung wird trivial erfüllt.

Damit haben wir unsere Funktion:

```
#include <stdint.h>

// extern uint32_t u32p32exp2m1( uint32_t );
#include "u32p32exp2m1.h"

int32_t
compute_ks( uint32_t f_hz, uint32_t th_us ){

        const uint64_t nominator = (1000ul * 1000ul) << 32;
        const uint64_t cs64 = nominator / f_hz / th_us;
        const uint32_t cs   = cs64 <= UINT32_MAX ? (uint32_t) cs64 : UINT32_MAX;
        const uint32_t ps   = u32p32exp2m1( cs );

        const uint64_t nom  = 1ul << 63;
        const uint64_t den  = ps + ( 1ul << 32 );

        const uint64_t ks64 = nom / den;
        const  int32_t ks   = ks64 <=  INT32_MAX ? ( int32_t) ks64 :  INT32_MAX;

        return ks;
}
```

Wir nutzen (die Konversionen erzeugen keinen Code und die konstanten Ausdrücke optimiert der Compiler weg):

- 1 Addition von oder Oderieren mit einer konstanten ganzzahligen Potenz von 2;
- 2 Triviale Vergleiche mit Auswahl;
- 3 Unsigned 64-Bit-Divisionen;
- 1 Aufruf von `u32p32exp2m1()`.

Divisionen und besonders der Aufruf von `u32p32exp2m1()` dominieren den Aufwand.

Wenn garantiert ist, dass das Produkt aus *f_hz* und *th_us* kleiner als `UINT32_MAX` ist, kann in der zweiten Zeile eine 64-Bit-Division eingespart werden:

```
        const uint64_t temp = nominator / ( f_hz * th_us );
```

7. Mathematisches Hilfsmittel

Ich mag self contained Dokumentation.

7.1. Geometrische Reihe

Kann man auch in einer Formelsammlung oder bei Wikipedia nachschauen, sich vom Video erklären lassen oder einfach wissen.

Zu berechnen ist:

(M.1) $S(n) = \sum_{i=0}^{n} k^i$

Wir entfernen das erste Element aus der Summe:

(M.2) $1 \cdot S(n) = 1 + \sum_{i=1}^{n} k^i$

Wir multiplizieren (M.1) mit k:

$$k \cdot S(n) = \sum_{i=0}^{n} k^{i+1}$$
$$k \cdot S(n) = \sum_{i=1}^{n+1} k^i$$

Und trennen das letzte Element von der Summe:

(M.3) $k \cdot S(n) = \sum_{i=1}^{n} k^i + k^{n+1}$

Wir subrahieren (M.3) von (M.2):

$$(1 - k) \cdot S(n) = 1 - k^{n+1}$$

(M.4) $S(n) = \frac{1 - k^{n+1}}{1 - k}$

Bei $|k| < 1$ gilt:

$$\lim_{n \to \infty} k^{n+1} = 0$$
$$\lim_{n \to \infty} S(n) = \frac{1 - \lim\limits_{n \to \infty} k^{n+1}}{1 - k}$$

(M.5) $\lim_{n \to \infty} S(n) = \frac{1}{1 - k}$

Wikipedia: Exponentielle Glättung [https://de.wikipedia.org/wiki/Exponentielle_Gl%C3%A4ttung].
