Liste des annexes

A Raspberry Pi 4 CPU info

Informations sur les CPU du Raspberry Pi 4 contenues dans le fichier /proc/cpuinfo.

```
$ cat /proc/cpuinfo
processor       : 0
model name      : ARMv7 Processor rev 3 (v7l)
BogoMIPS        : 108.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
 evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

processor       : 1
model name      : ARMv7 Processor rev 3 (v7l)
BogoMIPS        : 108.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
 evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

processor       : 2
model name      : ARMv7 Processor rev 3 (v7l)
BogoMIPS        : 108.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
 evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

processor       : 3
model name      : ARMv7 Processor rev 3 (v7l)
BogoMIPS        : 108.00
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4 idiva idivt vfpd32 lpae
 evtstrm crc32
CPU implementer : 0x41
CPU architecture: 7
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 3

Hardware        : BCM2711
Revision        : b03112
Serial          : 1000000022221a34
Model           : Raspberry Pi 4 Model B Rev 1.2
```

B CMake principal

Fichier CMake principal placé à la racine du projet. Il permet de compiler :

- le projet *audio_encoding* contenant les différentes MDCT et leurs tests : les commandes CMake de ce sous-projet sont présentées dans les annexes suivantes sous le code qu'elles permettent de compiler ;
- la librairie *Ne10* : les variables suivantes sont initialisées conformément aux recommendations de la documentation pour la compilation de la librairie :
  - NE10_LINUX_TARGET_ARCH est initialisée à armv7 (l'architecture du Raspberry Pi 4) ;
  - GNULINUX_PLATFORM est initialisée à ON ;
  - BUILD_DEBUG est initialisée à ON si le projet est compilé en mode *debug*.

```
cmake_minimum_required(VERSION 3.13)

set(NE10_LINUX_TARGET_ARCH armv7)
set(GNULINUX_PLATFORM ON)
if (CMAKE_BUILD_TYPE STREQUAL "DEBUG")
    set(BUILD_DEBUG ON)
endif (CMAKE_BUILD_TYPE STREQUAL "DEBUG")

add_subdirectory(audio_encoding)
add_subdirectory(Ne10)
```

C Valeurs constantes des MDCT

Le fichier `mdct_constants.h` rassemble les valeurs constantes des MDCT pour une fenêtre d'entrée de 1024 échantillons.

```
// Sampling frequency: 48kHz
#define FS                  48000

// Window length and derived constants
#define MDCT_WINDOW_LEN     1024
#define MDCT_M              (MDCT_WINDOW_LEN>>1)     // spectrum size
#define MDCT_M2             (MDCT_WINDOW_LEN>>2)     // fft size
#define MDCT_M4             (MDCT_WINDOW_LEN>>3)
#define MDCT_M32            (3*(MDCT_WINDOW_LEN>>2))
#define MDCT_M52            (5*(MDCT_WINDOW_LEN>>2))
```

D Algorithmes de référence

D.1 MDCT de référence en *floating point*

Algorithme de référence basé sur la formule mathématique de la MDCT. Le *template* permet de réaliser les calculs en *float* ou en *double*.

```
#include <cmath>

#include "mdct_constants.h"

template<typename FLOAT>
void ref_float_mdct(FLOAT *time_signal, FLOAT *spectrum)
{
    FLOAT scale = 2.0 / sqrt(MDCT_WINDOW_LEN);
    FLOAT factor1 = 2.0 * M_PI / static_cast<FLOAT>(MDCT_WINDOW_LEN);
    FLOAT factor2 = 0.5 + static_cast<FLOAT>(MDCT_M2);
    for (int k = 0; k < MDCT_M; ++k)
    {
        FLOAT result = 0.0;
        FLOAT factor3 = (k + 0.5) * factor1;
        for (int n = 0; n < MDCT_WINDOW_LEN; ++n)
        {
            result += time_signal[n] * cos((static_cast<FLOAT>(n) + factor2) * factor3);
        }
        spectrum[k] = scale * result;
    }
}
```

D.2 MDCT de référence en *fixed point*

Algorithme de référence basé sur la formule mathématique de la MDCT. Le spectre est calculé en *double* puis converti en *integer* sur 32 bits en représentation Q15.

```
#include <cassert>

#include "ref_mdct.h"

void ref_int_mdct(int16_t *time_signal, int32_t *spectrum)
{
    double scale = sqrt(MDCT_WINDOW_LEN) / 2.0;      // MDCT scale (2/sqrt(WIN_LEN)) + Q15 scale
    double factor1 = 2.0 * M_PI / MDCT_WINDOW_LEN;
    double factor2 = 0.5 + MDCT_M2;
    for (int k = 0; k < MDCT_M; ++k)
    {
        double result = 0.0;
        double factor3 = (k + 0.5) * factor1;
        for (int n = 0; n < MDCT_WINDOW_LEN; ++n)
        {
            result += time_signal[n] * cos((n + factor2) * factor3);
        }
        assert(round(result*scale) == static_cast<int32_t>(round(result*scale)));
        spectrum[k] = static_cast<int32_t>(round(result/scale));
    }
}
```

E Génération d'un signal sinusoïdal

E.1 Génération d'un signal sinusoïdal en *floating point*

Code de génération d'un signal sinusoïdal en *float* ou en *double*. Le *template* permet de générer ces deux types de signaux avec le même code.

```
#include <cmath>

template<typename FLOAT>
void sin_float(FLOAT *out, int n_samples, double amplitude,
    double frequency, double phase_shift, int sampling_frequency)
{
    FLOAT omega = 2.0 * M_PI * frequency / static_cast<FLOAT>(sampling_frequency);
    for (int i = 0; i < n_samples; ++i)
    {
        out[i] = amplitude * sin(static_cast<FLOAT>(i) * omega + phase_shift);
    }
}
```

E.2 Génération d'un signal sinusoïdal en *fixed point*

La génération du signal sinusoïdal en *integer* fait appel à la génération du signal sinusoïdal en *double* avant de convertir le résultat en *integer* (représentation Q15).

```
#include <cstring>

#include "sin_wave.h"

void sin_int(int16_t *out, int n_samples, double amplitude,
    double frequency, double phase_shift, int sampling_frequency)
{
    double scale = 1.0;
    if (abs(amplitude) < 1.0) scale *= amplitude;

    double *temp_sin = static_cast<double *>(malloc(n_samples*sizeof(double)));
    memset(temp_sin, 0, n_samples*sizeof(double));

    sin_float<double>(temp_sin, n_samples, scale, frequency, phase_shift, sampling_frequency);

    for (int i = 0; i < n_samples; ++i)
    {
        out[i] = static_cast<int16_t>(temp_sin[i]*pow(2.0, 15.0));
    }
}
```

F Implémentation de la MDCT basée sur la FFT de *FFTW3*

F.1 Header

Header de la classe `mdct_fftw3_f32` : MDCT basée sur la FFT de la librairie *FFTW3* en *float* (32 bits). La classe contient les structures de données `fft_in` et `fft_out`, le tableau de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`fft_plan`). L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#include <fftw3.h>

#include "mdct_constants.h"

class fftw3_mdct_f32
{
    private:
    fftwf_plan fft_plan;       // FFT configuration
    fftwf_complex *fft_in;     // FFT input buffer
    fftwf_complex *fft_out;    // FFT output buffer
    float twiddle[MDCT_M];

    public:
    fftw3_mdct_f32();
    ~fftw3_mdct_f32();
    void mdct(float *time_signal, float *spectrum);
    void imdct(float *spectrum, float *time_signal);
};
```

F.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_fftw3_f32` :

- Le tableau de `twiddle` est initialisé en *float* sur 32 bits ;
- La FFT de *FFTW3* est initialisée en une dimension (pour l'audio) avec la taille de la FFT réduite à un quart de la taille de la fenêtre d'entrée par le pre-processing et avec l'option `FFTW_MEASURE` plus lente à l'initialisation mais qui permet d'optimiser le temps d'exécution de la FFT ;
- Les tableaux contenant les données d'entrée (`fft_in`) et de sortie (`fft_out`) de la FFT sont alloués dynamiquement avec la fonction de *FFTW3* et ils sont passés en paramètre à la configuration de la FFT.

```
#include <cmath>

fftw3_mdct_f32::fftw3_mdct_f32()
{
    float alpha = M_PI / (8.f * MDCT_M);
    float omega = M_PI / MDCT_M;
    float scale = sqrt(sqrt(2.f / MDCT_M));

    for (int i = 0; i < MDCT_M2; ++i)
    {
        float x = omega*i + alpha;
        twiddle[2*i] = scale * cos(x);
        twiddle[2*i+1] = scale * sin(x);
    }
```

```
    fft_in = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);
    fft_out = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);
    fft_plan = fftwf_plan_dft_1d(MDCT_M2, fft_in, fft_out, FFTW_FORWARD, FFTW_MEASURE);
}
```

F.3 Destructeur

Destructeur de la classe `mdct_fftw3_f32` qui permet de libérer la mémoire allouée aux tableaux d'entrée et de sortie de la FFT et à sa configuration avec les fonctions appropriées fournies par la librairie *FFTW3*.

```
fftw3_mdct_f32::~fftw3_mdct_f32()
{
    fftwf_destroy_plan(fft_plan);
    fftwf_free(fft_in);
    fftwf_free(fft_out);
}
```

F.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *FFTW3* :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettent de réduire la fenêtre d'entrée de la FFT ;
- Appel de la fonction FFT de *FFTW3* ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettent de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle*.

```
void fftw3_mdct_f32::mdct(float *time_signal, float *spectrum)
{
    float *cos_tw = twiddle;
    float *sin_tw = cos_tw + 1;

    /* odd/even folding and pre-twiddle */
    float *xr = (float *)fft_in;
    float *xi = xr + 1;

    for (int i = 0; i < MDCT_M2; i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] + time_signal[MDCT_M32+i];
        float i0 = time_signal[MDCT_M2+i] - time_signal[MDCT_M2-1-i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        xr[i] = r0*c + i0*s;
        xi[i] = i0*c - r0*s;
    }

    for(int i = MDCT_M2; i < MDCT_M; i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] - time_signal[-MDCT_M2+i];
        float i0 = time_signal[MDCT_M2+i] + time_signal[MDCT_M52-1-i];
```

```
        float c = cos_tw[i];
        float s = sin_tw[i];

        xr[i] = r0*c + i0*s;
        xi[i] = i0*c - r0*s;
    }

    /* complex FFT of size MDCT_M2 */
    fftwf_execute(fft_plan);

    /* post-twiddle */
    xr = (float *)fft_out;
    xi = xr + 1;

    for (int i = 0; i < MDCT_M; i += 2)
    {
        float r0 = xr[i];
        float i0 = xi[i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        spectrum[i] = -r0*c - i0*s;
        spectrum[MDCT_M-1-i] = -r0*s + i0*c;
    }
}
```

G Validation de la MDCT *FFTW3* en *float 32*

G.1 Code source

Test de la MDCT basée sur la FFT de *FFTW3* en *float 32* avec un signal d'entrée sinusoïdal à 200 Hz :

- Génération et affichage d'un signal sinusoïdal à 200 Hz ;
- Calcul et affichage du spectre de fréquences de ce signal ;
- Opération inverse de la MDCT et affichage du signal temporel calculé à partir du spectre.

Les différentes données sont écrites dans un fichier CSV afin de pouvoir les exploiter sous forme graphique.

```
#include <iomanip>
#include <iostream>
#include <fstream>

#include <cstring>

#include "mdct_constants.h"
#include "fftw3_mdct_f32.h"
#include "sin_wave.h"

/**
    * @brief MDCT algorithm calling the FFT of the fftw3 library
    * Code based on https://www.dsprelated.com/showcode/196.php
    * Change the MDCT_WINDOW_LEN to test the MDCT with other spectrum sizes
    */
int main(void)
{
    // input time signal (200 Hz)
    float time_in[MDCT_WINDOW_LEN];
    sin_float(time_in, MDCT_WINDOW_LEN, 0.9, 200.0, 0.0, FS);

    // output time signal (generated by the IMDCT)
    float time_out[MDCT_WINDOW_LEN];
    memset(time_out, 0, MDCT_WINDOW_LEN*sizeof(float));

    // frequency spectrum (generated by the MDCT)
    float spectrum[MDCT_M];
    memset(spectrum, 0, MDCT_M*sizeof(float));

    // perform the MDCT and IMDCT
    fftw3_mdct_f32 fftw3_mdct;
    fftw3_mdct.mdct(time_in, spectrum);
    fftw3_mdct.imdct(spectrum, time_out);

    // print the results in CSV file
    std::ofstream csv_file;
    csv_file.open("fftw3_mdct_f32.csv");
    csv_file << "time (ms),signal in,signal out,frequency (Hz),spectrum" << std::endl;

    for (int i = 0; i < MDCT_M; ++i)
    {
        csv_file << i*1000.0/FS << "," << time_in[i] << "," << time_out[i] << ","
            << (i+1.0)*FS/MDCT_WINDOW_LEN << "," << std::abs(spectrum[i]) << "," << std::endl;
    }
```

```
    for (int i = MDCT_M; i < MDCT_WINDOW_LEN; ++i)
    {
        csv_file << i*1000.0/(FS) << "," << time_in[i] << "," << time_out[i] << "," << std::endl;
    }

    csv_file.close();

    return 0;
}
```

G.2 Compilation

Commandes CMake permettant de compiler le code d'exemple.

```
# MDCT using the fftw3 library f32
add_executable(fftw3_mdct_f32 test/validation/fftw3_example.cpp
    src/fftw3_mdct_f32.cpp src/sin_wave.cpp)
target_link_libraries(fftw3_mdct_f32 fftw3f)
```

H Implémentation de la MDCT basée sur la FFT de *Ne10* en *floating point*

H.1 Header

Header de la classe `mdct_ne10_f32_c` : MDCT basée sur la FFT de la librairie *Ne10* en *float* (32 bits). La classe contient les structures de données `fft_in` et `fft_out`, le tableau de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`cfg`). L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#pragma once

#include "mdct_constants.h"
#include "NE10.h"

class ne10_mdct_f32_c
{
    private:
    ne10_fft_cfg_float32_t cfg;                                                    // Ne10 configuration
    ne10_fft_cpx_float32_t fft_in[MDCT_M2] __attribute__((aligned(16)));   // Ne10 FFT input buffer
    ne10_fft_cpx_float32_t fft_out[MDCT_M2] __attribute__((aligned(16)));  // Ne10 FFT output buffer
    float twiddle[MDCT_M] __attribute__((aligned(16)));                            // twiddle factors

    public:
    ne10_mdct_f32_c();
    ~ne10_mdct_f32_c();
    void mdct(float *time_signal, float *spectrum);
};
```

H.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_ne10_f32_c` :

- Le tableau de `twiddle` est initialisé en *float* sur 32 bits ;
- La configuration de la FFT de *Ne10* est initialisée en *complex to complex* en *float 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée.

```
ne10_mdct_f32_c::ne10_mdct_f32_c()
{
    float alpha = M_PI / (8.0 * static_cast<float>(MDCT_M));
    float omega = M_PI / static_cast<float>(MDCT_M);
    float scale = sqrt(sqrt(2.0 / static_cast<float>(MDCT_M)));
    for (int i = 0; i < MDCT_M2; ++i)
    {
        float x = omega * i + alpha;
        twiddle[2*i] = static_cast<float>(scale * cos(x));
        twiddle[2*i+1] = static_cast<float>(scale * sin(x));
    }

    cfg = ne10_fft_alloc_c2c_float32_c(MDCT_M2);
}
```

H.3 Destructeur

Destructeur de la classe `mdct_ne10_f32_c` qui permet de libérer la mémoire allouée à la configuration de la FFT avec la fonction appropriée de la librairie *Ne10*.

```
ne10_mdct_f32_c::~ne10_mdct_f32_c()
{
    ne10_fft_destroy_c2c_float32(cfg);
}
```

H.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *Ne10* en *float 32* et en *plain C* :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettent de réduire la fenêtre d'entrée de la FFT ;
- Appel de la fonction FFT de *Ne10* ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettent de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle*.

```
void ne10_mdct_f32_c::mdct(float *time_signal, float *spectrum)
{
    // pre-twiddling
    float *cos_tw = twiddle;
    float *sin_tw = cos_tw + 1;
    for (int i = 0; i < MDCT_M2; i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] + time_signal[MDCT_M32+i];
        float i0 = time_signal[MDCT_M2+i] - time_signal[MDCT_M2-1-i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        fft_in[i/2].r = r0*c + i0*s;
        fft_in[i/2].i = i0*c - r0*s;
    }

    for (int i = MDCT_M2; i < (MDCT_M); i += 2)
    {
        float r0 = time_signal[MDCT_M32-1-i] - time_signal[-MDCT_M2+i];
        float i0 = time_signal[MDCT_M2+i] + time_signal[MDCT_M52-1-i];

        float c = cos_tw[i];
        float s = sin_tw[i];

        fft_in[i/2].r = r0*c + i0*s;
        fft_in[i/2].i = i0*c - r0*s;
    }

    // FFT
    ne10_fft_c2c_1d_float32_c(fft_out, fft_in, cfg, 0);
```

```
    // post-twiddling
    for (int i = 0; i < (MDCT_M); i += 2)
    {
        float r0 = fft_out[i/2].r;
        float i0 = fft_out[i/2].i;

        float c = cos_tw[i];
        float s = sin_tw[i];

        spectrum[i] = -r0*c - i0*s;
        spectrum[(MDCT_M)-1-i] = -r0*s + i0*c;
    }
}
```

I Mesure des performances des FFT de *Ne10*

I.1 Code source

Code permettant de tester la vitesse d'exécution moyenne de différentes FFT proposées par la librairie *Ne10*. La moyenne est calculée sur 10 000 000 d'exécutions. Les données d'entrée de la FFT sont générées aléatoirement et sont différentes pour chaque exécution. Les variables de préprocesseur définies à la compilation permettent à partir du même code de mesurer le temps d'exécution moyen avec écart type :

- de la FFT *complex to complex* en *float 32* en *plain C* ou avec les optimisations Neon ;
- de la FFT *complex to complex* en *integer 32* en *plain C* ou avec les optimisations Neon ;
- de la FFT *complex to complex* en *integer 16* en *plain C* ou avec les optimisations Neon.

```
#include <iomanip>
#include <iostream>
#include <limits>

#include <cmath>
#include <cstring>

#include "mdct_constants.h"
#include "Timers.h"
#include "NE10.h"

#ifdef F32          // 32 bits floating point arithmetic

#define INPUT_RANGE             1.8
#define INPUT_DATA              ne10_fft_cpx_float32_t
#define OUTPUT_DATA             ne10_fft_cpx_float32_t
#define FFT_CONFIG              ne10_fft_cfg_float32_t
#define DESTROY_CONFIG          ne10_fft_destroy_c2c_float32

#ifdef NEON
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_float32_neon
#define PERFORM_FFT             ne10_fft_c2c_1d_float32_neon
#else
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_float32_c
#define PERFORM_FFT             ne10_fft_c2c_1d_float32_c
#endif

#elif I32           // 32 bits fixed point arithmetic

#define INPUT_RANGE             std::numeric_limits<int16_t>::max()*2
#define INPUT_DATA              ne10_fft_cpx_int32_t
#define OUTPUT_DATA             ne10_fft_cpx_int32_t
#define FFT_CONFIG              ne10_fft_cfg_int32_t
#define DESTROY_CONFIG          ne10_fft_destroy_c2c_int32

#ifdef NEON
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_int32_neon
#define PERFORM_FFT             ne10_fft_c2c_1d_int32_neon
#else
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_int32_c
#define PERFORM_FFT             ne10_fft_c2c_1d_int32_c
#endif
```

```
#else                // 16 bits fixed point arithmetic

#define INPUT_RANGE             std::numeric_limits<int16_t>::max()*2
#define INPUT_DATA              ne10_fft_cpx_int16_t
#define OUTPUT_DATA             ne10_fft_cpx_int16_t
#define FFT_CONFIG              ne10_fft_cfg_int16_t
#define ALLOC_CONFIG            ne10_fft_alloc_c2c_int16
#define DESTROY_CONFIG          ne10_fft_destroy_c2c_int16

#ifdef NEON
#define PERFORM_FFT             ne10_fft_c2c_1d_int16_neon
#else
#define PERFORM_FFT             ne10_fft_c2c_1d_int16_c
#endif

#endif

#define RUNS            10000000
#define FFT_SCALE_FLAG  0

int main()
{
    // print which FFT will be tested
#ifdef F32
#ifdef NEON
    std::cout << "FFT Ne10 f32 Neon" << std::endl;
#else
    std::cout << "FFT Ne10 f32 plain C" << std::endl;
#endif

#elif I32
#ifdef NEON
    std::cout << "FFT Ne10 i32 Neon" << std::endl;
#else
    std::cout << "FFT Ne10 i32 plain C" << std::endl;
#endif
#else
#ifdef NEON
    std::cout << "FFT Ne10 i16 Neon" << std::endl;
#else
    std::cout << "FFT Ne10 i16 plain C" << std::endl;
#endif
#endif

    // seed the random
    srand(static_cast<unsigned>(time(0)));

    // initialize the configuration
    FFT_CONFIG cfg = ALLOC_CONFIG(MDCT_M2);

    // start the loop executing the FFTs
    int64_t *runtimes = static_cast<int64_t *>(malloc(RUNS * sizeof(int64_t)));
    for (int i = 0; i < RUNS; ++i)
    {
        // initialize an empty spectrum
        OUTPUT_DATA spectrum[MDCT_M2] __attribute__((aligned(16)));
        memset(&spectrum, 0, (MDCT_M2)*sizeof(OUTPUT_DATA));
```

```
        // generate random input data
        INPUT_DATA time_signal[MDCT_M2] __attribute__((aligned(16)));
        for (int i = 0; i < MDCT_M2; ++i)
        {
            time_signal[i].r = INPUT_RANGE * rand() / RAND_MAX - INPUT_RANGE / 2;
            time_signal[i].i = INPUT_RANGE * rand() / RAND_MAX - INPUT_RANGE / 2;
        }

        // perfom the FFT and measure the run time
        EvsHwLGPL::CTimers timer;
        timer.Start();
#ifdef F32
        PERFORM_FFT(time_signal, spectrum, cfg, 0);
#else
        PERFORM_FFT(time_signal, spectrum, cfg, 0, FFT_SCALE_FLAG);
#endif
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }

    // clean
    DESTROY_CONFIG(cfg);

    // compute the average
    double avg = 0.0;
    for (int i = 0; i < RUNS; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(RUNS);
    std::cout << "average run time:    " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (int i = 0; i < RUNS; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(RUNS);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << " ns" << std::endl;

    return 0;
}
```

I.2 Compilation

Commandes CMake utilisées pour générer les exécutables permettant de mesurer le temps d'exécution de différentes FFT proposées par la librairie *Ne10*. En fonction des variables de préprocesseur définies, les exéctables suivants sont générés :

- `run_fft_f32_c` est généré si la variable `F32` est définie pour mesurer le temps d'exécution de la FFT *float 32 plain C* ;
- `run_fft_i32_c` est généré si la variable `I32` est définie pour mesurer le temps d'exécution de la FFT *integer 32 plain C* ;
- `run_fft_i16_c` est généré par défaut pour mesurer le temps d'exécution de la FFT *integer 16 plain C* ;
- `run_fft_f32_neon` est généré si les variables `F32` et `NEON` sont définies pour mesurer le temps d'exécution de la FFT *float 32* avec optimisations *Neon* ;
- `run_fft_i32_neon` est généré si les variables `I32` et `NEON` sont définies pour mesurer le temps d'exécution de la FFT *integer 32* avec optimisations *Neon* ;

- `run_fft_i16_neon` est généré si la variable NEON est définie pour mesurer le temps d'exécution de la FFT *integer 16* avec optimisations *Neon*.

```
# Ne10 FFT performance (float32 plain C)
add_executable(run_ne10_fft_f32_c test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_f32_c PUBLIC -DF32)
target_link_libraries(run_ne10_fft_f32_c NE10)

# Ne10 FFT performance (int32 plain C)
add_executable(run_ne10_fft_i32_c test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i32_c PUBLIC -DI32)
target_link_libraries(run_ne10_fft_i32_c NE10)

# Ne10 FFT performance (int16 plain C)
add_executable(run_ne10_fft_i16_c test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i16_c PUBLIC -DI16)
target_link_libraries(run_ne10_fft_i16_c NE10)

# Ne10 FFT performance (float32 with neon optimizations)
add_executable(run_ne10_fft_f32_neon test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_f32_neon PUBLIC -DF32 -DNEON)
target_link_libraries(run_ne10_fft_f32_neon NE10)

# Ne10 FFT performance (int32 with neon optimizations)
add_executable(run_ne10_fft_i32_neon test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i32_neon PUBLIC -DI32 -DNEON)
target_link_libraries(run_ne10_fft_i32_neon NE10)

# Ne10 FFT performance (int16 with neon optimizations)
add_executable(run_ne10_fft_i16_neon test/performance/run_ne10_fft.cpp src/Timers.cpp)
target_compile_definitions(run_ne10_fft_i16_neon PUBLIC -DI16 -DNEON)
target_link_libraries(run_ne10_fft_i16_neon NE10)
```

J Mesure des performances des FFT de *FFTW3*

J.1 Code source

Code permettant de tester la vitesse d'exécution moyenne de la FFT en *float 32* de la librairie *FFTW3*. La moyenne est calculée sur 10 000 000 d'exécutions. Les données d'entrée de la FFT sont générées aléatoirement et sont différentes pour chaque exécution.

```
#include <iomanip>
#include <iostream>
#include <cmath>

#include <fftw3.h>

#include "mdct_constants.h"
#include "Timers.h"

#define RUNS 10000000

int main()
{
    // print which FFT will be tested
    std::cout << "FFT FFTW3 f32 plain C" << std::endl;

    // seed the random
    srand(static_cast<unsigned>(time(0)));

    // start the loop executing the FFTs
    int64_t *runtimes = static_cast<int64_t *>(malloc(RUNS * sizeof(int64_t)));
    for (int i = 0; i < RUNS; ++i)
    {
        // initialize an empty spectrum
        fftwf_complex *fft_out = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);

        // generate random input data
        fftwf_complex *fft_in = (fftwf_complex *)fftwf_malloc(sizeof(fftwf_complex) * MDCT_M2);
        float *x = (float *)fft_in;
        for (int i = 0; i < MDCT_M2; ++i)
        {
            x[i] = 1.8f * rand() / RAND_MAX - 1.8f / 2.0f;
        }

        // initialize the configuration
        fftwf_plan fft_plan = fftwf_plan_dft_1d(MDCT_M2, fft_in, fft_out,
            FFTW_FORWARD, FFTW_MEASURE);

        // perfom the FFT and measure the run time
        EvsHwLGPL::CTimers timer;
        timer.Start();
        fftwf_execute(fft_plan);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();

        // clean
        fftwf_destroy_plan(fft_plan);
        fftwf_free(fft_in);
        fftwf_free(fft_out);
    }
```

```
    // compute the average
    double avg = 0.0;
    for (int i = 0; i < RUNS; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(RUNS);
    std::cout << "average run time:    " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (int i = 0; i < RUNS; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(RUNS);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << " ns" << std::endl;

    return 0;
}
```

J.2 Compilation

Commandes CMake utilisées pour générer l'exécutable permettant de mesurer le temps d'exécution de la FFT *float 32* de la librairie *FFTW3*.

```
# FFTW3 FFT performance (float32)
add_executable(run_fftw3_fft_f32 test/performance/run_fftw3_fft_f32.cpp src/Timers.cpp)
target_link_libraries(run_fftw3_fft_f32 fftw3f)
```

K Implémentation de la MDCT basée sur la FFT de *Ne10* en *fixed point*

K.1 Header

Header de la classe `mdct_ne10_i32_c` : MDCT basée sur la FFT de la librairie *Ne10* en *integer* (32 bits). La classe contient les structures de données `fft_in` en représentation Q1.15 et `fft_out` en Q9.15, le tableau de facteurs de `twiddle` utilisé pour le pre- et le post-processing et la configuration de la FFT (`cfg`). L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#pragma once

#include "mdct_constants.h"
#include "NE10.h"

class ne10_mdct_i32_c
{
    private:
    ne10_fft_cfg_int32_t cfg;                                          // Ne10 configuration
    ne10_fft_cpx_int32_t fft_in[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT input buffer
                                                                       //   Q1.15
    ne10_fft_cpx_int32_t fft_out[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT output buffer
                                                                        //   Q9.15
    int16_t twiddle[MDCT_M] __attribute__((aligned(16)));              // MDCT twiddle factors

    public:
    ne10_mdct_i32_c();
    ~ne10_mdct_i32_c();
    void mdct(int16_t *time_signal, int32_t *spectrum);
};
```

K.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_ne10_i32_c` :

- Le tableau de `twiddle` est initialisé en *double* puis converti en *integer* (représentation Q15) ;
- La configuration de la FFT de *Ne10* est initialisée en *complex to complex* en *integer 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée.

```
ne10_mdct_i32_c::ne10_mdct_i32_c()
{
    // intialize the twiddling factors
    double alpha = M_PI / (8.0*MDCT_M);
    double omega = M_PI / MDCT_M;
    double scale = sqrt(sqrt(2.0 / static_cast<double>MDCT_M));
    for (int i = 0; i < MDCT_M2; ++i)
    {
        double x = omega * i + alpha;
        twiddle[2*i] = static_cast<int16_t>(cos(x)*scale*pow(2.0, 15.0));
        twiddle[2*i+1] = static_cast<int16_t>(sin(x)*scale*pow(2.0, 15.0));
    }

    // initialize the Ne10 FFT configuration
    cfg = ne10_fft_alloc_c2c_int32_c(MDCT_M2);
}
```

K.3 Destructeur

Destructeur de la classe `mdct_ne10_i32_c` qui permet de libérer la mémoire allouée à la configuration de la FFT avec la fonction appropriée de la librairie *Ne10*.

```
ne10_mdct_i32_c::~ne10_mdct_i32_c()
{
    ne10_fft_destroy_c2c_int32(cfg);
}
```

K.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *Ne10* en *integer 32* et en *plain C* :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettant de réduire la fenêtre d'entrée de la FFT sont faites en algorithmique *fixed point* ;
- Appel de la fonction FFT de *Ne10* ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettant de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle* sont faites en algorithmique *fixed point*.

```
void ne10_mdct_i32_c::mdct(int16_t *time_signal, int32_t *spectrum)
{
    // pre-twiddling
    // fft_in = (Q1.15 + Q1.15) * Q1.15/4 + (Q1.15 + Q1.15) * Q1.15/4
    //          1/4 Q1.30 + 1/4 Q1.30 + 1/4 Q1.30 + 1/4 Q1.30 -> Q1.30
    //          >>7 -> Q1.23 + 8 bits reserved for the FFT
    int16_t *cos_tw = twiddle;
    int16_t *sin_tw = cos_tw + 1;
    for (int i = 0; i < MDCT_M2; i += 2)
    {
        int32_t r0 = static_cast<int32_t>(time_signal[MDCT_M32-1-i]) + time_signal[MDCT_M32+i];
        int32_t i0 = static_cast<int32_t>(time_signal[MDCT_M2+i]) - time_signal[MDCT_M2-1-i];

        int16_t c = cos_tw[i];
        int16_t s = sin_tw[i];

        fft_in[i/2].r = (((r0*c)+64)>>7) + (((i0*s)+64)>>7);
        fft_in[i/2].i = (((i0*c)+64)>>7) - (((r0*s)+64)>>7);
    }

    for (int i = MDCT_M2; i < MDCT_M; i += 2)
    {
        int32_t r0 = static_cast<int32_t>(time_signal[MDCT_M32-1-i]) - time_signal[-MDCT_M2+i];
        int32_t i0 = static_cast<int32_t>(time_signal[MDCT_M2+i]) + time_signal[MDCT_M52-1-i];

        int16_t c = cos_tw[i];
        int16_t s = sin_tw[i];

        fft_in[i/2].r = (((r0*c)+64)>>7) + (((i0*s)+64)>>7);
        fft_in[i/2].i = (((i0*c)+64)>>7) - (((r0*s)+64)>>7);
    }

    // perform the FFT
    ne10_fft_c2c_1d_int32_c(fft_out, fft_in, cfg, 0, 0);
```

```
    // post-twiddling
    // spectrum = Q9.23>>8 * Q1.15/4 + Q9.23>>8 * Q1.15/4
    //          = Q9.15 * Q1.15/4 + Q9.15 * Q1.15/4
    //          = Q10.30/4 + Q10.30/4
    //          = Q11.30/4
    //          = Q9.30 >> 15 = Q9.15
    for (int i = 0; i < MDCT_M; i += 2)
    {
        int32_t r0 = fft_out[i/2].r;
        int32_t i0 = fft_out[i/2].i;

        int16_t c = cos_tw[i];
        int16_t s = sin_tw[i];

        spectrum[i] = (((-(static_cast<int64_t>(r0)+128)>>8)*c+16384)>>15)
            - ((((static_cast<int64_t>(i0)+128)>>8)*s+16384)>>15);
        spectrum[MDCT_M-1-i] = ((((-static_cast<int64_t>(r0)+128)>>8)*s+16384)>>15)
            + ((((static_cast<int64_t>(i0)+128)>>8)*c+16384)>>15);
    }
}
```

L Implémentation de la MDCT basée sur la FFT de *Ne10* en *fixed point* avec optimisations *Neon*

Le code de cette annexe utilise les instructions intrinsèques Neon et doit être compilé avec l'option `-mfpu=neon`.

L.1 Header

Header de la classe `mdct_ne10_i32_neon` : MDCT basée sur la FFT de la librairie *Ne10* en *integer* (32 bits) optimisée par l'utilisation des opérations SIMD Neon. La classe contient les structures de données `fft_in` en représentation Q1.15 et `fft_out` en Q9.15, les tableaux de facteurs de `twiddle` utilisés pour le *pre-* et le *post-processing* et la configuration de la FFT (`cfg`). Contrairement aux autres implémentations, les facteurs de twiddle ne sont pas rassemblés dans un seul tableau. Les tableaux de facteurs de *pre-twiddling* et de *post-twiddling* sont séparés car ils sont utilisés en 16 bits pour le *pre-twiddling* et en 32 bits pour le *post-twiddling*. Chacun de ces tableaux est séparé en deux afin que chaque moitié puisse être initialisée dans un ordre qui facilite l'utilisation des opérations SIMD. L'implémentation des fonctions de ce header est présentée dans les annexes suivantes.

```
#pragma once

#include <arm_neon.h>
#include "mdct_constants.h"
#include "NE10.h"

class ne10_mdct_i32_neon
{
private:
    ne10_fft_cfg_int32_t cfg;                                          // Ne10 configuration
    ne10_fft_cpx_int32_t fft_in[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT input buffer
    ne10_fft_cpx_int32_t fft_out[MDCT_M2] __attribute__((aligned(16))); // Ne10 FFT output buffer
    int16_t pretwiddle_start[MDCT_M2] __attribute__((aligned(16)));    // pre-twiddle factors
    int16_t pretwiddle_end[MDCT_M2] __attribute__((aligned(16)));      // second half is stored
                                                                       //   in reversed order
    int32_t posttwiddle_start[MDCT_M2] __attribute__((aligned(16)));   // post-twiddle factors
    int32_t posttwiddle_end[MDCT_M2] __attribute__((aligned(16)));     // second half is stored
                                                                       //   in reversed order

public:
    ne10_mdct_i32_neon();
    ~ne10_mdct_i32_neon();
    void mdct(int16_t *time_signal, int32_t *spectrum);
};
```

L.2 Constructeur

Initialisation de la MDCT dans le constructeur de la classe `mdct_ne10_i32_neon` :

- Le tableau de `twiddle` est initialisé en *double* puis converti en *integer* (représentation Q15 en 16 bits pour le *pre-twiddling* et en 32 bits pour le *post-twiddling*) : la première moitié des tableaux est rangée à l'endroit dans les tableaux `pretwiddle_start` et `posttwiddle_start` tandis que la seconde est rangée à l'envers dans les tableaux `pretwiddle_end` et `posttwiddle_end` ;
- La configuration de la FFT de *Ne10* est initialisée en *complex to complex* en *integer 32* avec en paramètre la taille de la fenêtre de la FFT réduite à un quart de la taille de la fenêtre d'entrée avec la fonction adaptée pour l'exécution d'une FFT optimisée avec les instructions SIMD Neon.

```
ne10_mdct_i32_neon::ne10_mdct_i32_neon()
{
    double alpha = M_PI / (8.0*MDCT_M);
    double omega = M_PI / MDCT_M;
    double scale = sqrt(sqrt(2.0 / static_cast<double>MDCT_M));
    for (int i = 0; i < MDCT_M4; ++i)
    {
        double start = omega * (i) + alpha;
        double end = omega * (i+MDCT_M4) + alpha;
        double cos_start = cos(start);
        double sin_start = sin(start);
        double cos_end = cos(end);
        double sin_end = sin(end);
        pretwiddle_start[2*i] = static_cast<int16_t>(cos_start*scale*pow(2.0, 15.0));
        pretwiddle_start[2*i+1] = static_cast<int16_t>(sin_start*scale*pow(2.0, 15.0));
        pretwiddle_end[MDCT_M2-2*i-2] = static_cast<int16_t>(cos_end*scale*pow(2.0, 15.0));
        pretwiddle_end[MDCT_M2-2*i-1] = static_cast<int16_t>(sin_end*scale*pow(2.0, 15.0));
        posttwiddle_start[2*i] = static_cast<int32_t>(cos_start*scale*pow(2.0, 31.0));
        posttwiddle_start[2*i+1] = static_cast<int32_t>(sin_start*scale*pow(2.0, 31.0));
        posttwiddle_end[MDCT_M2-2*i-2] = static_cast<int32_t>(cos_end*scale*pow(2.0, 31.0));
        posttwiddle_end[MDCT_M2-2*i-1] = static_cast<int32_t>(sin_end*scale*pow(2.0, 31.0));
    }

    cfg = ne10_fft_alloc_c2c_int32_neon(MDCT_M2);
}
```

L.3 Destructeur

Destructeur de la classe `mdct_ne10_i32_c` qui permet de libérer la mémoire allouée à la configuration de la FFT avec la fonction appropriée de la librairie *Ne10*.

```
ne10_mdct_i32_neon::~ne10_mdct_i32_neon()
{
    ne10_fft_destroy_c2c_int32(cfg);
}
```

L.4 Fonction MDCT

Implémentation de l'algorithme de MDCT basé sur la FFT de la librairie *Ne10* en *integer 32* avec utilisation des instructions SIMD Neon :

- Initialisation du tableau d'entrée de la FFT : les opérations de *pre-twiddling* permettant de réduire la fenêtre d'entrée de la FFT sont faites en algorithmique *fixed point*. L'utilisation des fonctions SIMD permet d'effectuer quatre opérations en parallèle afin de réduire le temps d'exécution. Les facteurs de *pre-twiddling* codés sur 16 bits transforment le signal d'entrée codé sur *16 bits* en un tableau d'entrée de la FFT codé sur 32 bits ;
- Appel de la fonction FFT de *Ne10* optimisée par l'utilisation des instructions SIMD Neon ;
- Calcul du spectre de fréquences : les opérations de *post-twiddling* permettant de calculer le spectre à partir des données de sortie de la FFT et des facteurs de *twiddle* sont faites en algorithmique *fixed point*. Les fonctions SIMD permettent d'effectuer deux ou quatre opérations en parallèle afin de réduire le temps d'exécution.

```
void ne10_mdct_i32_neon::mdct(int16_t *time_signal, int32_t *spectrum)
{
    // see the twiddling_loops.nlsx file for more details

    // for i from 0 to 254, step 2

    // r[  0 -> 127, pas 1] =  time_signal[ 767 ->  513, pas 2] * c[  0 -> 127, pas 1]
    //                       +  time_signal[ 768 -> 1022, pas 2] * c[  0 -> 127, pas 1]
    //                       +  time_signal[ 256 ->  510, pas 2] * s[  0 -> 127, pas 1]
    //                       + -time_signal[ 255 ->    1, pas 2] * s[  0 -> 127, pas 1]

    // i[  0 -> 127, pas 1] =  time_signal[ 256 ->  510, pas 2] * c[  0 -> 127, pas 1]
    //                       + -time_signal[ 255 ->    1, pas 2] * c[  0 -> 127, pas 1]
    //                       + -time_signal[ 767 ->  513, pas 2] * s[  0 -> 127, pas 1]
    //                       + -time_signal[ 768 -> 1022, pas 2] * s[  0 -> 127, pas 1]

    // fft_in[i/2].r = time_signal[M32-1-i]    * cos_tw[i] + time_signal[M32+i]     * cos_tw[i]
    //               + time_signal[M2+i]       * sin_tw[i] + (-time_signal[M2-1-i]) * sin_tw[i]

    // fft_in[i/2].i = time_signal[M2+i]       * cos_tw[i] + (-time_signal[M2-1-i]) * cos_tw[i]
    //               + (-time_signal[M32-1-i]) * sin_tw[i] + (-time_signal[M32+i])  * sin_tw[i]


    // for i from 510 to 256, step 2

    // r[255 -> 128, pas 1] =  time_signal[ 257 ->  511, pas 2] * c[255 -> 128, pas 1]
    //                       + -time_signal[ 254 ->    0, pas 2] * c[255 -> 128, pas 1]
    //                       +  time_signal[ 766 ->  512, pas 2] * s[255 -> 128, pas 1]
    //                       +  time_signal[ 769 -> 1023, pas 2] * s[255 -> 128, pas 1]

    // i[255 -> 128, pas 1] =  time_signal[ 766 ->  512, pas 2] * c[255 -> 128, pas 1]
    //                       +  time_signal[ 769 -> 1023, pas 2] * c[255 -> 128, pas 1]
    //                       + -time_signal[ 257 ->  511, pas 2] * s[255 -> 128, pas 1]
    //                       +  time_signal[ 254 ->    0, pas 2] * s[255 -> 128, pas 1]

    // fft_in[i/2].r = time_signal[M32-1-i]    * cos_tw[i] + (-time_signal[-M2+i]) * cos_tw[i]
    //               + time_signal[M2+i]       * sin_tw[i] + time_signal[M52-1-i]  * sin_tw[i]

    // fft_in[i/2].i = time_signal[M2+i]       * cos_tw[i] + time_signal[M52-1-i] * cos_tw[i]
    //               + (-time_signal[M32-1-i]) * sin_tw[i] + time_signal[-M2+i]   * sin_tw[i]

    for (int i = 0; i < MDCT_M2; i += 8)
    {
        // tx.val[0] -> odd indexes
        // tx.val[1] -> even indexes
        int16x4x2_t t1 = vld2_s16(time_signal+MDCT_M2+i);
        int16x4x2_t t2 = vld2_s16(time_signal+MDCT_M2-8-i);
        int16x4x2_t t3 = vld2_s16(time_signal+MDCT_M32+i);
        int16x4x2_t t4 = vld2_s16(time_signal+MDCT_M32-8-i);

        t2.val[0] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t2.val[0]));
                                            // reverse the t2 even values: 0 2 4 6 -> 6 4 2 0
        t2.val[1] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t2.val[1]));
                                            // reverse the t2 odd values : 1 3 5 7 -> 7 5 3 1
        t4.val[0] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t4.val[0]));
                                            // reverse the t4 even values: 0 2 4 6 -> 6 4 2 0
        t4.val[1] = (int16x4_t)vrev64_s32((int32x2_t)vrev32_s16(t4.val[1]));
                                            // reverse the t4 odd values : 1 3 5 7 -> 7 5 3 1
```

```
        // x_tw.val[0] -> cos twiddle
        // x_tw.val[1] -> sin twiddle
        int16x4x2_t start_tw = vld2_s16(pretwiddle_start+i);
        int16x4x2_t end_tw = vld2_s16(pretwiddle_end+i);

        // start.val[0] -> real part
        // start.val[1] -> imaginary part
        int32x4x2_t start;
        start.val[0] = vshrq_n_s32(
            vaddq_s32(
                vaddq_s32
                    vmull_s16(t4.val[1], start_tw.val[0]),
                    vmull_s16(t3.val[0], start_tw.val[0])),
                vaddq_s32
                    vmull_s16(t1.val[0], start_tw.val[1]),
                    vmull_s16(vneg_s16(t2.val[1]), start_tw.val[1])))
                , 7);
        start.val[1] = vshrq_n_s32
            vaddq_s32(
                vaddq_s32
                    vmull_s16(t1.val[0], start_tw.val[0]),
                    vmull_s16(vneg_s16(t2.val[1]), start_tw.val[0])),
                vaddq_s32
                    vmull_s16(vneg_s16(t4.val[1]), start_tw.val[1]),
                    vmull_s16(vneg_s16(t3.val[0]), start_tw.val[1])))
                , 7);

        // end.val[0] -> real part
        // end.val[1] -> imaginary part
        int32x4x2_t end;
        end.val[0] = vshrq_n_s32
            vaddq_s32(
                vaddq_s32(vmull_s16(t1.val[1], end_tw.val[0]),
                vmull_s16(vneg_s16(t2.val[0]), end_tw.val[0])),
            vaddq_s32
                vmull_s16(t4.val[0], end_tw.val[1]),
                vmull_s16(t3.val[1], end_tw.val[1])))
            , 7);
        end.val[1] = vshrq_n_s32
            vaddq_s32(
                vaddq_s32
                    vmull_s16(t4.val[0], end_tw.val[0]),
                    vmull_s16(t3.val[1], end_tw.val[0])),
            vaddq_s32
                vmull_s16(vneg_s16(t1.val[1]), end_tw.val[1]),
                vmull_s16(t2.val[0], end_tw.val[1])))
            , 7);

        // reverse the end part
        end.val[0] = (int32x4_t)vrev64q_s32(end.val[0]);
        end.val[0] = vcombine_s32(vget_high_s32(end.val[0]), vget_low_s32(end.val[0]));
        end.val[1] = (int32x4_t)vrev64q_s32(end.val[1]);
        end.val[1] = vcombine_s32(vget_high_s32(end.val[1]), vget_low_s32(end.val[1]));

        // store the result
        vst2q_s32((int32_t *)fft_in+i, start);
        vst2q_s32((int32_t *)(fft_in+MDCT_M2-4-i/2), end);
    }
```

```
    // perform the FFT
    ne10_fft_c2c_1d_int32_neon(fft_out, fft_in, cfg, 0, 0);

    // post-twiddling
    for (int i = 0; i < MDCT_M2; i += 8)
    {
        // load the fft ouput and reverse the end part
        // fft_out_x.val[0] -> real part
        // fft_out_x.val[1] -> imaginary part
        int32x4x2_t fft_out_start = vld2q_s32((int32_t *)fft_out+i);
        int32x4x2_t fft_out_end = vld2q_s32((int32_t *)fft_out+MDCT_M-8-i);
        fft_out_end.val[0] = (int32x4_t)vrev64q_s32(fft_out_end.val[0]);
        fft_out_end.val[0] = vcombine_s32(vget_high_s32(fft_out_end.val[0]),
                                          vget_low_s32(fft_out_end.val[0]));
        fft_out_end.val[1] = (int32x4_t)vrev64q_s32(fft_out_end.val[1]);
        fft_out_end.val[1] = vcombine_s32(vget_high_s32(fft_out_end.val[1]),
                                          vget_low_s32(fft_out_end.val[1]));

        // load the twiddle factors
        // x_tw.val[0] -> cos twiddle
        // x_tw.val[1] -> sin twiddle
        int32x4x2_t start_tw = vld2q_s32(posttwiddle_start+i);
        int32x4x2_t end_tw = vld2q_s32(posttwiddle_end+i);

        int32x4x2_t spectrum_start;
        spectrum_start.val[0] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_start.val[0]), start_tw.val[0]),
            vqrdmulhq_s32(vnegq_s32(fft_out_start.val[1]), start_tw.val[1])), 8);
        spectrum_start.val[1] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_end.val[0]), end_tw.val[1]),
            vqrdmulhq_s32(fft_out_end.val[1], end_tw.val[0])), 8);

        int32x4x2_t spectrum_end;
        spectrum_end.val[0] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_end.val[0]), end_tw.val[0]),
            vqrdmulhq_s32(vnegq_s32(fft_out_end.val[1]), end_tw.val[1])), 8);
        spectrum_end.val[1] = vshrq_n_s32(vaddq_s32(
            vqrdmulhq_s32(vnegq_s32(fft_out_start.val[0]), start_tw.val[1]),
            vqrdmulhq_s32(fft_out_start.val[1], start_tw.val[0])), 8);

        spectrum_end.val[0] = (int32x4_t)vrev64q_s32(spectrum_end.val[0]);
        spectrum_end.val[0] = vcombine_s32(vget_high_s32(spectrum_end.val[0]),
                                           vget_low_s32(spectrum_end.val[0]));
        spectrum_end.val[1] = (int32x4_t)vrev64q_s32(spectrum_end.val[1]);
        spectrum_end.val[1] = vcombine_s32(vget_high_s32(spectrum_end.val[1]),
                                           vget_low_s32(spectrum_end.val[1]));

        // store the result
        vst2q_s32((int32_t *)spectrum+i, spectrum_start);
        vst2q_s32((int32_t *)(spectrum+MDCT_M-8-i), spectrum_end);
    }
}
```

M Validation des spectres de fréquences produits par les MDCT

M.1 Code source

M.2 Compilation

N Mesure des performances des MDCT *Ne10* et MDCT de référence

N.1 Code source

Code permettant de mesurer les performances des MDCT de *Ne10* en *float 32 plain C*, *integer 32 plain C*, *integer 32* avec optimisation Neon et de la MDCT de référence en *double* (en fonction de la variable de préprocesseur définie à la compilation). Le code mesure et affiche le temps d'exécution moyen et l'écart type. Ces informations sont calculées sur un nombre d'exécution donné en paramètre à l'exécutable. Les MDCT sont testées avec le même signal sinusoïdal en entrée dont la valeur par défaut est de 200 Hz.

```
#include <iostream>
#include <cstring>

#include "args_parser.h"
#include "mdct_constants.h"
#include "sin_wave.h"
#include "Timers.h"


#ifdef FIXED_POINT_C     // fixed point arithmetic

#include "ne10_mdct_i32_c.h"

#define INPUT_DATA              int16_t
#define OUTPUT_DATA             int32_t
#define GENERATE_SIN            sin_int
#define MDCT                    ne10_mdct_i32_c

#elif FIXED_POINT_NEON   // fixed point arithmetic

#include "ne10_mdct_i32_neon.h"

#define INPUT_DATA              int16_t
#define OUTPUT_DATA             int32_t
#define GENERATE_SIN            sin_int
#define MDCT                    ne10_mdct_i32_neon

#elif FLOATING_POINT     // floating point arithmetic

#include "ne10_mdct_f32_c.h"

#define INPUT_DATA              float
#define OUTPUT_DATA             float
#define GENERATE_SIN            sin_float
#define MDCT                    ne10_mdct_f32_c

#else                    // reference algorithm in floating point arithmetic

#include "ref_mdct.h"

#define INPUT_DATA              double
#define OUTPUT_DATA             double
#define GENERATE_SIN            sin_float

#endif
```

```
/**
 * @brief Run the MDCT on a single frame x times
 * the signal is a single tone configurable via the --sin parameter (200Hz by default)
 * the number of runs is setted by the --run parameter (1 by default)
 */
int main(int argc, char **argv)
{
    // initialize the parameters
    params p;
    try
    {
        p = parse_args(argc, argv);
    }
    catch(const std::runtime_error &err)
    {
        std::cerr << err.what() << std::endl;
        usage();
        return 1;
    }

    // print which MDCT will be tested
#ifdef FIXED_POINT_C
    std::cout << "MDCT Ne10 i32 plain C: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#elif FIXED_POINT_NEON
    std::cout << "MDCT Ne10 i32 Neon: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#elif FLOATING_POINT
    std::cout << "MDCT Ne10 f32 plain C: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#else
    std::cout << "Reference MDCT: "
        << p.runs << " runs with a single tone signal (" << p.frequency << "Hz)" << std::endl;
#endif

    // generate the time signal
    INPUT_DATA time_signal[MDCT_WINDOW_LEN] __attribute__((aligned(16)));
    memset(&time_signal, 0, MDCT_WINDOW_LEN*sizeof(INPUT_DATA));
    GENERATE_SIN(time_signal, MDCT_WINDOW_LEN, 0.9, p.frequency, 0.0, FS);

    // initialize an empty spectrum
    OUTPUT_DATA mdct_spectrum[MDCT_M] __attribute__((aligned(16)));
    memset(&mdct_spectrum, 0, MDCT_M*sizeof(OUTPUT_DATA));

    // perform the MDCT x times
    EvsHwLGPL::CTimers timer;
    int64_t *runtimes = static_cast<int64_t *>(malloc(p.runs * sizeof(int64_t)));

#ifdef REF
    for (unsigned i = 0; i < p.runs; ++i) {
        timer.Start();
        ref_float_mdct<INPUT_DATA>(time_signal, mdct_spectrum);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }
```

```
#else
    MDCT ne10_mdct;
    for (unsigned i = 0; i < p.runs; ++i) {
        timer.Start();
        ne10_mdct.mdct(time_signal, mdct_spectrum);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }
#endif

    // compute the average run time
    double avg = 0.0;
    for (unsigned i = 0; i < p.runs; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(p.runs);
    std::cout << "average run time: " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (unsigned i = 0; i < p.runs; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(p.runs);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << std::endl;

    // clean
    free(runtimes);

    return 0;
}
```

N.2 Compilation

Commandes CMake pour la compilation des différents exécutables de tests de performance des MDCT :

- la variable `FLOATING_POINT` permet de compiler l'exécutable `run_mdct_f32_c` pour tester la MDCT basée sur la FFT de *Ne10* en *float 32 plain C* ;
- la variable `FIXED_POINT_C` permet de compiler l'exécutable `run_mdct_i32_c` pour tester la MDCT basée sur la FFT de *Ne10* en *integer 32 plain C* ;
- la variable `FIXED_POINT_NEON` permet de compiler l'exécutable `run_mdct_i32_neon` pour tester la MDCT basée sur la FFT de *Ne10* en *integer 32* avec optimisations Neon ;
- la variable `REF` permet de compiler l'exécutable `run_mdct_ref` pour tester la MDCT de référence en *double* ;

```
# Ne10 f32 x times on a single frame
add_executable(run_ne10_mdct_f32_c test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ne10_mdct_f32_c.cpp)
target_compile_definitions(run_ne10_mdct_f32_c PUBLIC -DFLOATING_POINT)
target_link_libraries(run_ne10_mdct_f32_c NE10)

# Ne10 i32 plain C x times on a single frame
add_executable(run_ne10_mdct_i32_c test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ne10_mdct_i32_c.cpp)
target_compile_definitions(run_ne10_mdct_i32_c PUBLIC -DFIXED_POINT_C)
target_link_libraries(run_ne10_mdct_i32_c NE10)
```

```
# Ne10 i32 neon x times on a single frame
add_executable(run_ne10_mdct_i32_neon test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ne10_mdct_i32_neon.cpp)
target_compile_definitions(run_ne10_mdct_i32_neon PUBLIC -DFIXED_POINT_NEON)
target_link_libraries(run_ne10_mdct_i32_neon NE10)

# Ref MDCT x times on a single frame
add_executable(run_ref_mdct test/performance/run_ne10_mdct.cpp
    src/args_parser src/sin_wave.cpp src/Timers.cpp
    src/ref_mdct.cpp)
target_compile_definitions(run_ref_mdct PUBLIC -DREF)
target_link_libraries(run_ref_mdct NE10)
```

O Mesure des performances de la MDCT *FFTW3* en *float 32*

O.1 Code source

Code permettant de mesurer les performances de la MDCT basée sur la FFT de *FFTW3* en *float 32*. Le code mesure et affiche le temps d'exécution moyen et l'écart type. Ces informations sont calculées sur 10 000 000 d'exécutions. La MDCT est testée avec le même signal sinusoïdal en entrée dont la valeur est définie à 440 Hz.

```
#include <iostream>
#include <cstring>

#include "fftw3_mdct_f32.h"
#include "sin_wave.h"
#include "Timers.h"

#define RUNS        10000000
#define FREQUENCY   440.0

int main()
{
    // print which MDCT will be tested
    std::cout << "MDCT FFTW3 f32 plain C: "
        << RUNS << " runs with a single tone signal (" << FREQUENCY << "Hz)" << std::endl;

    // generate the time signal
    float time_signal[MDCT_WINDOW_LEN];
    sin_float(time_signal, MDCT_WINDOW_LEN, 0.9, FREQUENCY, 0.0, FS);

    // initialize an empty spectrum
    float spectrum[MDCT_M];
    memset(spectrum, 0, MDCT_M*sizeof(float));

    // initialize the configuration
    fftw3_mdct_f32 fftw3_mdct;

    // perform the MDCT
    EvsHwLGPL::CTimers timer;
    int64_t *runtimes = static_cast<int64_t *>(malloc(RUNS * sizeof(int64_t)));
    for (int i = 0; i < RUNS; ++i)
    {
        timer.Start();
        fftw3_mdct.mdct(time_signal, spectrum);
        timer.Stop();
        runtimes[i] = timer.GetTimeElapsed();
    }

    // compute the average run time
    double avg = 0.0;
    for (int i = 0; i < RUNS; ++i) avg += static_cast<double>(runtimes[i]);
    avg = avg / static_cast<double>(RUNS);
    std::cout << "average run time: " << avg << " ns" << std::endl;

    // compute the standard deviation
    double dev = 0.0;
    for (int i = 0; i < RUNS; ++i) dev += static_cast<double>(runtimes[i]) - avg;
    dev = dev * dev / static_cast<double>(RUNS);
    dev = sqrt(dev);
    std::cout << "standard deviation: " << dev << std::endl;
```

```
    // clean
    free(runtimes);

    return 0;
}
```

O.2 Compilation

Commandes CMake pour la compilation de l'exécutable de tests de performance de la MDCT basée sur la FFT de *FFTW3* en *float 32*.

```
# FFTW3 f32
add_executable(run_fftw3_mdct_f32 test/performance/run_fftw3_mdct_f32.cpp
    src/fftw3_mdct_f32.cpp src/sin_wave.cpp src/Timers.cpp)
target_link_libraries(run_fftw3_mdct_f32 fftw3f)
```