Elaborazione Sequenziale di Segnali Audio con DFT e IDFT

Contents

1 Introduzione

Questo progetto si propone di elaborare segnali audio utilizzando un approccio sequenziale. L'obiettivo è implementare trasformazioni nel dominio delle frequenze per applicare un filtro passa-basso e produrre un segnale audio modificato. Il processo comprende le seguenti fasi:

- Lettura di un file audio in formato `.wav`;
- Calcolo della Trasformata Discreta di Fourier (DFT);
- Applicazione di un filtro passa-basso nel dominio delle frequenze;
- Calcolo della Trasformata Inversa (IDFT);
- Scrittura del segnale audio modificato su file;
- Generazione di un report sui tempi di esecuzione.

1.1 Fase di test

Tutti i codici sviluppati verranno testati su due macchine differenti per apprezzare la differenza tra architetture delle CPU e lo scarto generazionale delle GPU. Le macchine in questione hanno le seguenti specifiche:

	Macchina 1	**Macchina 2**
CPU	Intel Core ?? ??	AMD Ryzen 7 8845HS
GPU	NVIDIA GeForce 30?? ??	NVIDIA GeForce RTX 4050 Max-Q
O.S.	Ubuntu ?? on wsl	Fedora Linux 41

2 Implementazione Sequenziale

Il codice è stato scritto in linguaggio `C` e utilizza le librerie standard per la manipolazione di file e la misurazione del tempo. La struttura principale del programma comprende:

2.1 Lettura e Scrittura di File Audio

I file audio sono letti e scritti in formato `.wav` standard, utilizzando un'intestazione di 44 byte per rappresentare i metadati del file. Le seguenti funzioni implementano la lettura e scrittura dei file audio:

Listing 1: Funzione per leggere file audio .wav.

```c
// Funzione per leggere i campioni audio da un file .wav
void readWavFile(const char *filename, double *x, int N) {
    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
        printf("Errore nell'apertura del file %s\n", filename);
        exit(1);
    }

    // Salta l'intestazione del file .wav (44 byte standard)
    fseek(file, 44, SEEK_SET);

    // Leggi i campioni audio come int16_t e convertili in double
    int16_t *buffer = (int16_t *)malloc(N * sizeof(int16_t));
    fread(buffer, sizeof(int16_t), N, file);
    for (int i = 0; i < N; i++) {
        x[i] = (double)buffer[i];
    }

    free(buffer);
    fclose(file);
}
```

Listing 2: Funzione per scrivere file audio .wav.

```c
// Funzione per scrivere un file .wav con l'intestazione
void writeWavFile(const char *filename, double *x, int N) {
    FILE *file = fopen(filename, "wb");
    if (file == NULL) {
        printf("Errore nell'apertura del file %s\n", filename);
        exit(1);
    }

```

```
    // Scrivi un'intestazione standard per un file .wav a 16 bit, mono, 44.1 kHz
    uint8_t header[44] = {
        'R', 'I', 'F', 'F',
        0, 0, 0, 0, // Placeholder per la dimensione del file
        'W', 'A', 'V', 'E',
        'f', 'm', 't', ' ',
        16, 0, 0, 0, // Dimensione del blocco fmt
        1, 0, // PCM
        1, 0, // Canali (mono)
        0x44, 0xAC, 0x00, 0x00, // Frequenza di campionamento: 44100 Hz
        0x88, 0x58, 0x01, 0x00, // Byte rate: 44100 * 2
        2, 0, // Block align: 2 byte
        16, 0, // Bit depth: 16 bit
        'd', 'a', 't', 'a',
        0, 0, 0, 0 // Placeholder per la dimensione dei dati
    };

    // Calcola la dimensione totale del file e dei dati
    int dataSize = N * sizeof(int16_t);
    int fileSize = 44 + dataSize - 8;

    // Aggiorna i campi dell'intestazione
    header[4] = (fileSize & 0xFF);
    header[5] = ((fileSize >> 8) & 0xFF);
    header[6] = ((fileSize >> 16) & 0xFF);
    header[7] = ((fileSize >> 24) & 0xFF);

    header[40] = (dataSize & 0xFF);
    header[41] = ((dataSize >> 8) & 0xFF);
    header[42] = ((dataSize >> 16) & 0xFF);
    header[43] = ((dataSize >> 24) & 0xFF);

    // Scrivi l'intestazione
    fwrite(header, sizeof(uint8_t), 44, file);

    // Scrivi i campioni audio convertiti in int16_t
    int16_t *buffer = (int16_t *)malloc(N * sizeof(int16_t));
    for (int i = 0; i < N; i++) {
        buffer[i] = (int16_t)x[i];
    }
    fwrite(buffer, sizeof(int16_t), N, file);

    free(buffer);
    fclose(file);
}
```

2.2 Trasformata Discreta di Fourier (DFT)

La DFT converte un segnale audio dal dominio del tempo a quello delle frequenze. La funzione `dft()` implementa la seguente formula:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot \cos\left(\frac{2\pi k n}{N}\right) \tag{1}$$

Listing 3: Implementazione della DFT.

```c
// Funzione che calcola la Discrete Fourier Transform di un segnale audio
void dft(double *x, double *X, int N) { // N = numero di campioni, complessit  O(N^2)
    for (int k = 0; k < N; k++) {
        X[k] = 0;
        for (int n = 0; n < N; n++) {
            X[k] += x[n] * cos(2 * PI * k * n / N);
        }
    }
}
```

2.3 Filtro Passa-Basso

Un filtro passa-basso è applicato nel dominio delle frequenze per rimuovere le componenti indesiderate. La funzione `filtro()` azzera le frequenze al di fuori di una soglia specificata:

Listing 4: Applicazione del filtro passa-basso.

```c
// Funzione che applica un filtro passa-basso al segnale audio
void filtro(double *X, int N, int fc) {
    for (int k = 0; k < N; k++) {
        if (k > fc && k < N - fc) {
            X[k] = 0;
        }
    }
}
```

2.4 Trasformata Inversa (IDFT)

La IDFT riporta il segnale audio dal dominio delle frequenze al dominio del tempo. La funzione `idft()` è basata sulla formula:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \cdot \cos\left(\frac{2\pi k n}{N}\right) \tag{2}$$

Listing 5: Implementazione della IDFT.

```c
// Funzione che calcola l'Inverse Discrete Fourier Transform del segnale audio filtrato
void idft(double *X, double *x, int N) { // N = numero di campioni, complessit  O(N^2)
    for (int n = 0; n < N; n++) {
        x[n] = 0;
        for (int k = 0; k < N; k++) {
            x[n] += X[k] * cos(2 * PI * k * n / N);
        }
        x[n] /= N;
    }
}
```

2.5 Misurazione delle Prestazioni

I tempi di esecuzione delle diverse fasi del programma vengono misurati e riportati in un file di testo:

Listing 6: Funzione per scrivere il report dei tempi.

```c
// Funzione per scrivere un report dei tempi di esecuzione
void writeReport(const char *filename, double dftTime, double
    filterTime, double idftTime, double totalTime) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        printf("Errore nell'apertura del file %s\n", filename);
        exit(1);
    }

    fprintf(file, "Report tempi di esecuzione:\n");
    fprintf(file, "-----------------------------------\n");
    fprintf(file, "DFT   : %f secondi\n", dftTime);
    fprintf(file, "Filtro: %f secondi\n", filterTime);
    fprintf(file, "IDFT  : %f secondi\n", idftTime);
    fprintf(file, "Totale: %f secondi\n", totalTime);
    fprintf(file, "-----------------------------------\n");
    fclose(file);
}
```

3 Esecuzione del Programma

La funzione `main()` coordina l'intero processo, dalla lettura del file audio alla scrittura dei risultati. Viene utilizzata la libreria `time.h` per misurare i tempi di esecuzione:

Listing 7: Funzione main del programma.

```c
// Funzione main, prende in input il nome del file audio e restituisce
    il file audio filtrato
int main(int argc, char *argv[] ) {
    double *x, *X, *y;
    int N;
    clock_t start, end;
    double dftTime, filterTime, idftTime;
    char *filename;

    // Verifica che il numero di argomenti sia corretto
    if (argc != 2) {
        printf("Utilizzo: %s <file_audio.wav>\n", argv[0]);
        exit(1);
    }

    filename = argv[1];

    // Creazione delle directory ./output e ./reports se non esistono
    mkdir("./output", 0777);
    mkdir("./reports", 0777);

    // Determina la lunghezza del file audio
    N = getWavFileLength(filename);

```

```
    // Allocazione dinamica della memoria
    x = (double *)malloc(N * sizeof(double));
    X = (double *)malloc(N * sizeof(double));
    y = (double *)malloc(N * sizeof(double));

    // Creazione di un timestamp
    char timestamp[20];
    createTimestamp(timestamp, sizeof(timestamp));

    // Percorsi per i file di output
    char outputFile[256], reportFile[256];
    snprintf(outputFile, sizeof(outputFile), "./output/output_%s.wav",
        timestamp);
    snprintf(reportFile, sizeof(reportFile), "./reports/report_%s.txt",
         timestamp);

    // Leggi il file audio
    readWavFile(filename, x, N);

    // Calcola la DFT
    start = clock();
    dft(x, X, N);
    end = clock();
    dftTime = (double)(end - start) / CLOCKS_PER_SEC;

    // Applica il filtro passa-basso
    start = clock();
    filtro(X, N, 1000);
    end = clock();
    filterTime = (double)(end - start) / CLOCKS_PER_SEC;

    // Calcola la IDFT
    start = clock();
    idft(X, y, N);
    end = clock();
    idftTime = (double)(end - start) / CLOCKS_PER_SEC;

    // Scrivi il file output
    writeWavFile(outputFile, y, N);

    // Scrivi il report dei tempi
    writeReport(reportFile, dftTime, filterTime, idftTime, dftTime +
        filterTime + idftTime);

    // Libera la memoria
    free(x);
    free(X);
    free(y);

    return 0;
}
```

4 Risultati

Il programma è stato testato su un file audio mono con 44.1 kHz di frequenza di campionamento e una durata di 0,5s e 5s .

I risultati della applicazione del filtro si trovano nella directory `./output`, ogni file generato dal programma possiede nel nome il timestamp dell'esecuzione nel formato Ymd_HMS (anno, mese, giorno, ore, minuti, secondi).

I tempi di esecuzione sono riportati nei file all'interndo della directory `./reports`. I risultati mostrano tempi di calcolo consistenti con le complessità delle operazioni.

Le seguenti tabelle mostrano i risultati ottenuti su entrambe le macchine e entrambi i file di input.