Relazione del progetto di Introduzione all'Audio Digiale: Audio Processing Tools

Federico Mariti

March 8, 2013

Lo scopo del progetto è la realizzazione di un programma che consenta l'elaborazione offline di file audio. Le trasformazioni studiate sono

- il cambiamento di volume,
- la normalizzazione,
- controllo di tono.

L'applicativo è realizzato in modo modulare così da permettere una facile estensione dell'insieme di trasformazioni messe a disposizione. Alcune operazioni di elaborazione audio, ad esempio la normalizzazione, richiedono la conoscenza di tutto il contenuto audio. Per facilitare l'implementazione si è deciso di copiare in memoria tutto il contenuto del file audio da elaborare. Il linguaggio di programmazione scelto è Java in quanto fronisce un'interfaccia audio omogenea rispetto a sistemi operativi diversi e consente di agire a basso livello sul contenuto audio, con il livello di astrazione dei dati byte dei campioni audio.

Una operazione di elaborazione audio è modellata come un modulo avente una funzione detta *filter* che assume come parametro un array di bytes, rappresentate l'audio da elaborare.

```
void filter(byte[] data, int off, int len)
```

Il modulo è inoltre caratterizzato da un formato audio necessario per interpretare correttamente i bytes audio all'interno della funzione filter.

L'interfaccia Java `Module` dichiara i metodi forniti da un modulo di elaborazione audio:

```
void filter(byte[] data, int off, int len)
AudioFormat getAudioFormat()
void setAudioFormat(AudioFormat af)
```

La classe astratta `AbstractModule` implementa `Module`, contiene un oggetto di tipo `javax.sound.sampled.AudioFormat` e implementa i metodi setter e getter per il fomato audio del modulo, il metodo *filter* rimane astratto e viene

implementato dalle classi *final* che realizzano lo specifico comportamento di moduli di elaborazione.

Sono state implementate due diverse interfacce utente: una a riga di comando e l'altra grafica. Indipendentemente dall'interfaccia utente le operazioni svolte dal programma sono le seguenti:

1. apertura di un file audio, selezionato dall'utente,
   (a) verifica che il formato audio sia supportato,
   (b) verifica che la dimensione del file sia completamente allocabile in memoria
2. creazione di un `AudioInputStream` dal file aperto,
3. mapping in memoria di tutto il contenuto del file,
4. creazione di un modulo di elaborazione audio che implementa la trasformazione richiesta dall'utente e esecuzione della sua funzione `filter` sul contenuto del file scritto in memoria,
5. scrittura della trasformazione in un file scelto dall'utente (ha lo stesso formato del file di ingresso).

1 Volume

Nello studio di elaborazioni audio il cambiamento di volume è la prima trasformazione da effettuare in quanto la sua realizzazione è molto semplice. Consiste nell'applicare, tramite moltiplicazione, un fattore costante a tutti i campioni di ogni canale dell'audio considerato. Così facendo si aumenta (se il fattore è maggiore di 1) o si riduce l'ampiezza del segnale audio considerato. La buona realizzazione di tale trasformazione è perciò delegata alla corretta interpretazione dei dati audio (dimensione dei campioni audio, ordinamento dei bytes). L'algoritmo è descritto dalla seguente formula in cui $x[n]$ definisce il segnale audio (di un canale) di lunghezza n campioni:

$$\alpha \in \mathbb{N}, \quad \forall i \in \{0, \ldots, n-1\}\colon\ y[i] = \alpha \cdot x[i]$$

L'implementazione Java della funzione *filter* fa uso di funzionalità di conversione da bytes a interi e viceversa contenute nella classe `AudioUtils`. L'oggetto `Volume` che implementa `Module` contiene l'oggetto intero `volume` che rappresenta il fattore da applicare al segnale audio.

```
for (i=off; i<len; i+=sampleSize) {
  if (2 == sampleSize)
    next = AudioUtils.byte2int(data[i], data[i+1],
        isBigEndian);
  else if (3 == sampleSize)
    next = AudioUtils.byte2int(data[i], data[i+1], data[i
        +2], isBigEndian);
```

```
    next = (int)(next * volume);
    byte[] tmp = AudioUtils.int2byte(next, isBigEndian);
    data[i] = tmp[0];
    data[i+1] = tmp[1];
    if (3 == sampleSize)
        data[i+2] = tmp[2];
}
```

Codice 1: La parte rilevante dell'implementazione della funzione filter della classe Volume

2 Normalizzazione

La normalizzazione di una informazione audio consiste nell'aumentare l'ampiezza del segnale audio in modo costante per tutta la durata del segnale così da renderla massima e non introdurre distorsione. Si tratta di un caso particolare di aumento di volume, il fattore di incremento è scelto analizzando tutto il segnale audio e portando il picco più alto al massimo valore rappresentabile dalla dimensione del campione. È possibile sceglire di normalizzare anche ad un valore inferiore a quello massimo.

L'implementazione è simile a quella del modulo *volume*, ne differisce per una prima scansione di tutto il contenuto audio, ricercando il valore massimo in valore assoluto dei campioni, senza fare distinzione tra i canali. Quindi il fattore di *gain* viene calcolato come:

```
int upperBound = 1 << af.getSampleSizeInBits();
if (af.getEncoding() == AudioFormat.Encoding.PCM_SIGNED)
    upperBound = upperBound / 2;
upperBound = upperBound - 1;
gain = upperBound * (targetLevel/100F) / maxPeak;
```

Dove `af` è l'oggetto che rappresenta il formato audio del segnale, `targetLevel` è il valore di normalizzazione scelto dall'utente e `maxPeak` è il valore del picco più alto trovato nel segnale audio. Successivamente viene eseguito lo stesso algoritmo di *volume* usando `gain` come valore di volume.

3 Controllo di tono

Per controllo di tono si intende una alterazione dello spettro di tutto il segnale audio considerato, tale elaborazione è una trasformazione lineare e invariante nel tempo. Il modulo implementato realizza un filtro digitale della famiglia FIR (Finite Inpulse Response). Filtri di questo tipo si basano sull'operazione matematica di convoluzione di due segnali, il filtro viene modellato come un sistema lineare e invariante nel tempo, e perciò descritto completamente dalla sua risposta alla funzione impulso. Il comportamento del filtro è una combinazione lineare di un gruppo consecutivo di campioni del segnale considerato. Sia $x[n]$

il segnale audio considerato composto da n campioni, si indica con $x[i]$ la componente i-esima del segnale x. La risposta impulsiva del filtro è un segnale di k componenti, in genere $n >> k$, la componente i-esima del filtro, anche detta coefficiente i-esimo del filtro, è indicata con c_i. Il comportamento del filtro è specificato dalla seguente relazione, con $y[n]$ il segnale risultato:

$$\forall i \in \{0, \ldots, n-1\}: \; y[i] = \sum_{j=0}^{k-1} c_j \cdot x[i-j]$$

Ovvero la componente i-esima del segnale risultato è una combinazione lineare (definita dai coefficenti del filtro) dei k componenti di x precedenti a $x[i]$. Il calcolo delle componenti di y può essere descritto con il seguente algoritmo:

$\forall i \in \{0, \ldots, n-1\}:$
    $y[i] := 0$
    $\forall j \in \{0, \ldots, k-1\}:$
        if $i - j > 0 \; \wedge \; i - j < n$ **then** $y[i] := y[i] + c_j \cdot x[i-j]$

La definizione del filtro e quindi dei suoi coefficienti viene effettuata con l'ausilio di un'applicazione esterna, ad esempio
`http://arc.id.au/FilterDesign.html`. Il programma richiede che l'utente specifichi un file testuale contenente i coefficenti del filtro, analizza il file e salva i coefficienti nell'oggetto che implementa il modulo.

Di seguito viene fornita l'implemetazione Java della funzione *filter* del modulo che realizza il comportamento di un FIR. Tale implementaizone segue l'algoritmo presentato precedentemente. All'interno di tale funzione viene usato un array di interi come array circolare per memorizzare i k campioni audio di x necessari al calcolo del campione attuale del risultato. Al calcolo di una generica componente $y[i]$, il valore di $x[i]$ viene decodificato dallo stream di bytes del segnale audio e scritto nella posizione corrente del buffer circolare, rimpiazzando così il valore più vecchio presente nel buffer.

```
//define: tail is the index of the next buffer element to
//        write, that is, tail is the index of the oldest
//        element in buffer
int tail = 0;
int[] buffer = new int[coef.length];
for (int i=off; i<len; i+=sampleSize) {
  // y[i] = 0
  result = 0;
  // x[i]
  if (2 == sampleSize)
    buffer[tail] = AudioUtils.byte2int(data[i], data[i+1],
        isBigEndian);
  else if (3 == sampleSize)
    buffer[tail] = AudioUtils.byte2int(data[i], data[i+1],
        data[i+2], isBigEndian);
```

```
    for (int j=0; j<coef.length; j++) {
      if (i-j > 0 && i-j < len) {
        // y[i] = y[i] + c[j] * x[i-j]
        int index;
        if (tail-j < 0) index = buffer.length + tail - j;
        else            index = tail-j;
        result += coef[j] * buffer[index];
      }
    }
    byte[] tmp = AudioUtils.int2bytes((int)result, isBigEndian
        );
    data[i] = tmp[0];
    data[i+1] = tmp[1];
    tail = (tail+1) % coef.length;
  }
```

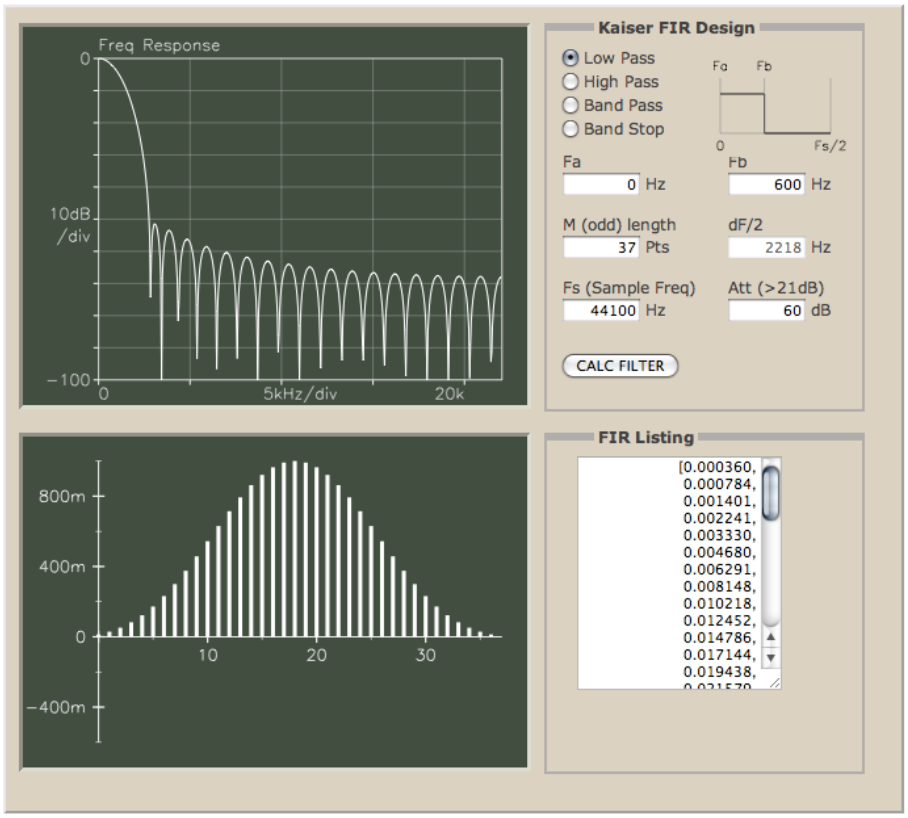

Figura 1: Schermata dell'applicativo la definizione del filtro e il calcolo dei relativi coefficenti

4 Esempi di uso

4.1 Command line interface

La stampa di aiuto del programma:

```
$ java mariti.audio.iad.CommandLineInterface -h
usage: -f inputFile -m moduleName [-a argName=value ...] -o outputFile
  -a,--argument
  -f,--file <fileName>        The input audio file
  -h,--help                   Shows the help content
  -m,--module <moduleName>    The audio processig module name
  -o,--output-file <fileName> The output audio file
Modules arguments:
  volume: -a volume=<integer>
  peakNormalization: -a peak=<integer>
  lowPass: nothing
  fir: -a file=<fileName>
  fir: -a coefficients=<float,float,...>
```

Esempio di uso del programma per la modifica di tono:

```
$ java mariti.audio.iad.CommandLineInterface -f "500 miles high.wav" \
  - o out.wav -m fir -a file=coef_lowPass_44100.txt
```

L'invocazione del programma richiede il file audio su cui effettuare l'elaborazione, il file audio di uscita contenente il risulato dell'elaborazione, e il modulo di elaborazione con i relativi argomenti. Una singola invocazione del programma produce una singola trasformazione.

4.2 Graphical user interface

Vengono presentate le stampe del programma con i diversi moduli di elaborazione. La finestra dell'applicazione è composta da 4 sezioni disposte lungo l'asse verticale:

- i controlli del pannello superiore consentono di definire il file audio su cui effettuare l'elaborazione,
- il pannello centrale è uno schedario che consente di scegliere quale modulo di elaborazione usare, all'interno dello schedario vengono presentati i componenti che definiscono i valori degli argomenti del modulo correntemente selezionato.
- i controlli del terzo pannello consentono di definire il file audio su cui scrivere l'elaborazione,
- il pannello più basso presenta il pulsante di scrittura, la pressione di tale pulsante aziona l'esecuzione dell'operazione specificata dai controlli nei pannelli superiori.

A differenza del programma a riga di comando è perciò possibile effettuare più elaborazioni all'interno di una singola esecuzione del programma.

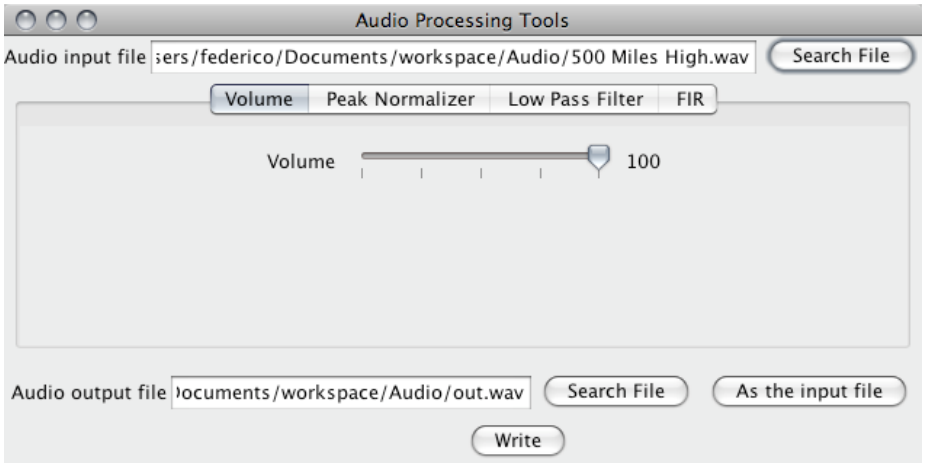

Figura 2: Schermata del modulo volume

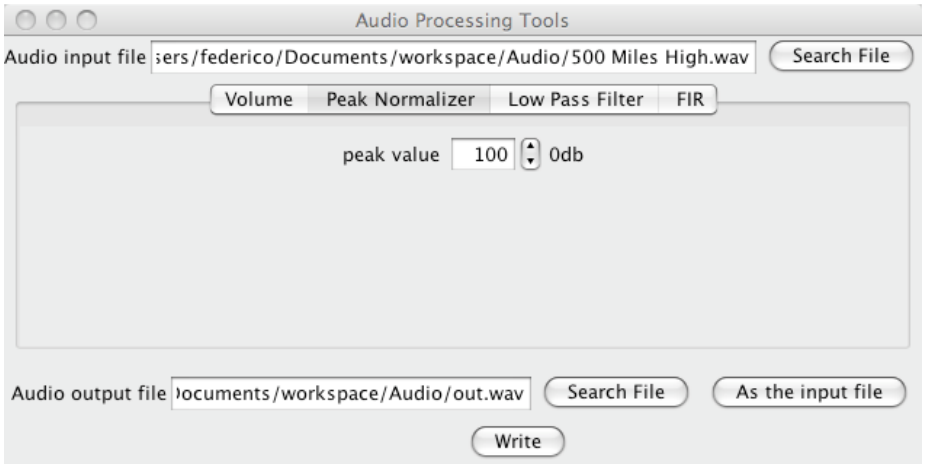

Figura 3: Schermata del modulo normalizzatore

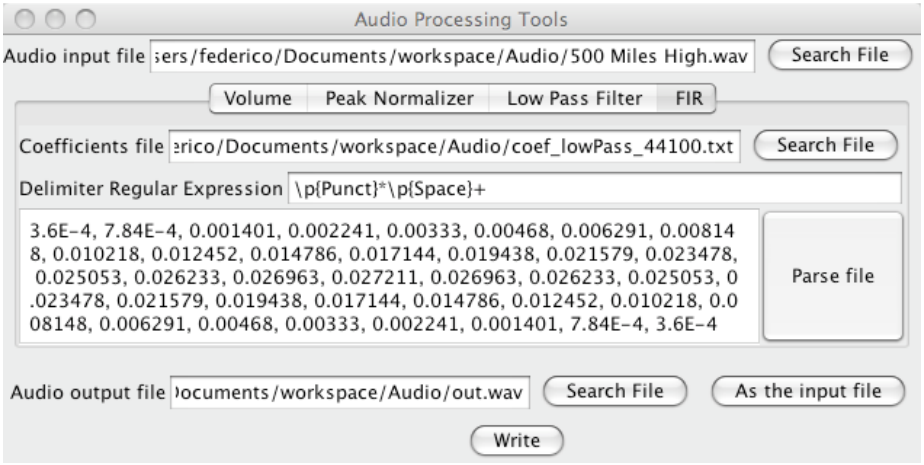

Figura 4: Schermata del modulo FIR

Nella scheda del modulo FIR viene chiesta la definizione di un file testuale contenente i coefficenti del filtro da applicare, tale file può avere un formato arbitrario ed è specificato da una espressione regolare. Una volta scelto il file dei coefficenti, il suo contenuto viene mostrato in una regione di testo modificabile con formato di delimitazione delle componenti “virgola e spazio”.