Autotune

Giacomo Radicchi

Maggio 2025

Indice

1 Introduzione

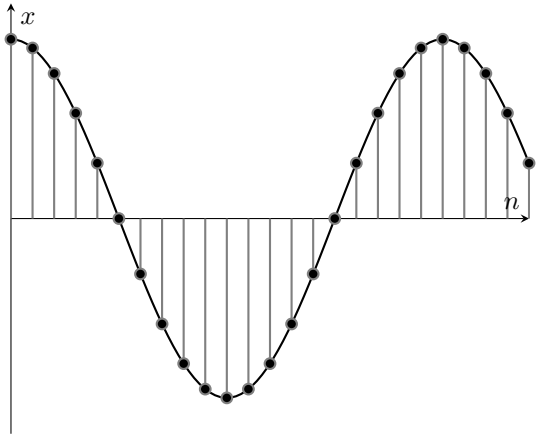
L'*autotune* è uno degli strumenti più utilizzati nella musica contemporanea. Nato alla fine degli anni '90, il suo impatto sulla produzione musicale è stato talmente profondo da trasformarsi in una vera e propria firma sonora, soprattutto nei generi pop, trap e hip-hop. La sua funzione principale è quella di correggere l'intonazione delle linee vocali, adattandole a una scala musicale di riferimento. In ogni istante, l'autotune rileva l'intonazione effettiva della voce e la modifica quel tanto che basta affinché corrisponda alla nota più vicina all'interno della scala selezionata.

Nel presente lavoro, si analizzeranno i fondamenti teorici alla base dell' elaborazione dell'intonazione, soffermandosi in particolare su tecniche come il time-stretching, il pitch-shifting, l'utilizzo della STFT e dell'autocorrelazione. Si affronterà anche un'implementazione in ambiente *MATLAB* e *Python* di alcuni algoritmi base utili alla realizzazione di un semplice sistema di autotune.

2 Cenni teorici

2.1 Concetto di pitch

Immaginiamo di avere come sequenza una sinusoide del tipo $x[n] = cos(2\pi f_o n)$



Nello specifico, f_o rappresenta la frequenza in Hertz (**Hz**) che determina l'intonazione, o **pitch**, del segnale. Può anche essere visto come il reciproco del periodo del segnale $T = \frac{1}{f_0}$, relazione che ci tornerà più utile in avanti.

2.2 Frequenza di una nota

Ad ogni nota musicale è associata la propria frequenza. In realtà, è necessario definirla solamente per una nota, tipicamente il La - o A in inglese -, per determinare tutte le altre 12 note della scala cromatica (Do, Do#, Re, Re#, Mi, Fa, Sol, Sol#, La, La#, Si). Questo perché per poter ricavare la frequenza della

nota successiva, vale a dire aumentare il pitch di un **semitono**, è necessario moltiplicarlo per $2^{\frac{1}{12}}$. Da notare che la scala delle note è periodica di periodo 12, perciò avanzare di 12 semitoni - dunque moltiplicare per 2 - equivale a prendere la stessa nota ma di un'**ottava** superiore. La frequenza standard attribuita al La è quella di **440** Hz e da questa si possono ricavare le rimanenti. Per esempio, se si volesse calcolare il pitch del Si (B), basta moltiplicare quello del La (A) per $2^{\frac{2}{12}}$, dunque 440 Hz $\times$ 1.122 = 493.883 Hz.

2.3 Discrete Fourier Transform

Come vedremo successivamente, è necessario riuscire a passare dal dominio del tempo a dominio della frequenza. Per fare ciò, si applica la **DFT** (Discrete Fourier Transform) la cui formula è la seguente:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi \frac{k}{N} n} \tag{1}$$

dove N è la dimensione della sequenza X[n]. Da notare che X[k] è ottenuta per campionamento di uno spettro periodico di periodo 1, nonché della **DTFT** (Discrete-Time Fourier Transform), dunque la DFT sarà anch'essa periodica ma di periodo N.

2.4 Short-Time Fourier Transform

La **STFT** (Short-Time Fourier Transform) ci da come informazione come varia la trasformata di Fourier al variare del tempo. In particolare, la sequenza originale viene suddivisa in una serie di sotto-sequenze, o frame, ottenute mediante il prodotto con una specifica window function (ad esempio: Hann, Hamming, Blackman, ecc.):

$$x_i[n] = x[n + i(N - L)] w_N[n] \tag{2}$$

dove:

- N è la lunghezza della finestra, o **frame length**
- L è il numero di campioni sovrapposti tra un frame e il successivo, ovvero l'**hop length**
- $x_i[n]$ è la **window function** di lunghezza N

Da notare che le sotto sequenze sono sovrapposte per evitare discontinuità nel calcolo della STFT.

Per ogni frame $x_i[n]$, viene calcolata la Discrete Fourier Transform:

$$X_i[k] = \sum_{n=0}^{N-1} x_i[n] e^{-j2\pi \frac{k}{N} n} \tag{3}$$

2.5 Autocorrelazione

Dato un segnale $x(t)$, l'autocorrelazione misura la somiglianza del segnale con se stesso al variare di un ritardo temporale τ:

$$R_{xx}^{(E)}(\tau) = \int_{-\infty}^{\infty} x^*(t)\, x(t+\tau)\ dt \tag{4}$$

Dove $x^*(t)$ rappresenta il complesso coniugato di x(t). Nel caso di segnali reali, in cui il complesso coniugato coincide con il segnale originale, la formula si semplifica in:

$$R_{xx}^{(E)}(\tau) = \int_{-\infty}^{\infty} x(t)\, x(t+\tau)\ dt \tag{5}$$

Nel caso dei segnali discreti, l'integrale si riduce ad una sommatoria:

$$R_{xx}^{(E)}[l] = \sum_{n=-\infty}^{\infty} x[n]\, x[n+l] \tag{6}$$

In pratica, l'autocorrelazione consente di evidenziare la periodicità presente nel segnale: per un segnale periodico, i valori di $R_{xx}^{(E)}(\tau)$ mostrano picchi nei ritardi corrispondenti al periodo fondamentale e ai suoi multipli. Questo rende l'autocorrelazione uno strumento efficace per stimare la frequenza fondamentale, o pitch, di una linea vocale.

3 Modifica del pitch

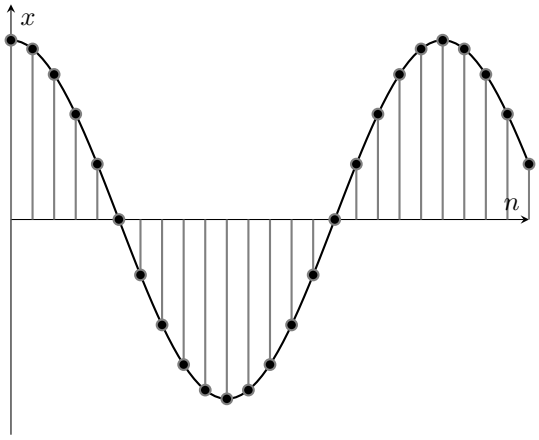
La modifica dell'intonazione di una traccia audio rappresenta una delle problematiche più discusse nella modellazione audio. Come è possibile riuscire ad aumentare la frequenza di un qualsiasi audio a piacimento?

3.1 Resampling dell'audio

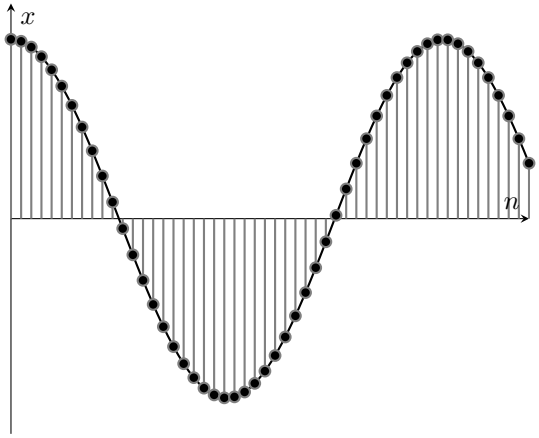
Un primo metodo che permette di fare ciò è quello di sovracampionare o sottocampionare il segnale, ovvero fare del **resampling**. Queste pratiche danno luogo, rispettivamente, ad una diminuzione o aumento del pitch.

Immaginiamo di voler dimezzare la frequenza della traccia audio, nonché abbassare il pitch di un'ottava. Per fare ciò, è necessario sovracampionare il nostro segnale dimezzando la distanza tra ogni singolo campione e, dunque, raddoppiare il numero di campioni presenti all'interno dell'audio.

Audio originale:



Audio sovracampionato:



Potremmo implementare ciò attraverso un semplice programma:

- MATLAB:

```
function x_out = resample(file_name, pitch_factor)
    [data, sr] = audioread(file_name);

    % lavoriamo con audio mono per semplicita'
    if size(data, 2) > 1
        data = data(:, 1);
    end

    og_length = size(data, 1);

    % genero l'asse dei tempi originale:
    old_t = 1 : og_length;
    % genero l'asse dei tempi del nuovo audio:
```

```
    new_t = 1 : pitch_factor : og_length;

    % nuova sequenza ottenuta per interpolazione lineare
        :
    x_out = interp1(old_t, data, new_t);

    % salvataggio audio:
    audiowrite("resampled_audio.wav", x_out, sr);

    x_out = data;
end
```

- Python:

```
import numpy as np
import soundfile as sf

audio, sr = sf.read("audio.wav")

# lavoriamo con audio mono per semplicita'
if len(np.shape(audio)) > 1:
   audio = audio[:, 0]

og_length = np.shape(audio)[0]
pitch_factor = 0.5

# genero l'asse dei tempi originale:
old_t = np.arange(0, og_length)
# genero l'asse dei tempi del nuovo audio:
new_t = np.arange(0, og_length, pitch_factor)

# nuova sequenza ottenuta per interpolazione lineare:
new_audio = np.interp(new_t, old_t, audio)

# salvataggio audio
sf.write("pitched_audio1.wav", new_audio, sr)
```

Nell'esempio fornito, i campioni sono stati interpolati linearmente. Questo tipo di ricostruzione non è del tutto preciso e può portare ad artefatti o suoni indesiderati. L'ideale sarebbe attuare un'interpolazione cardinale sfruttando la formula tratta dal teorema del campionamento di Shannon-Nyquist:

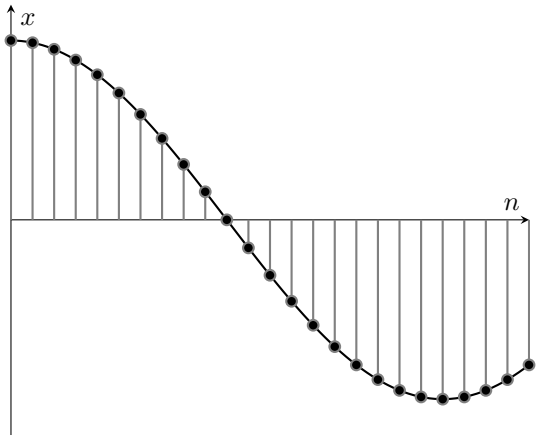
$$x(t) = \sum_{n=-\infty}^{\infty} x(nT_c)\text{sinc}\left[\frac{\pi}{T_c}(t - nT_c)\right] \tag{7}$$

Per questioni di efficienza, verranno utilizzate le funzioni "interp1" per MATLAB e "scipy.signal.resample" della libreria scipy per Python.

Ovviamente, la frequenza di campionamento, o **sample rate**, con la quale si andrà a salvare la nuova traccia sovracampionata dovrà rimanere la stessa

precedente, altrimenti l'audio riprodotto rimarrà totalmente identico o, nel caso di sottocampionamento, potrebbe risultare distorto a causa del fenomeno di *aliacing*.

Audio sovracampionato con sample rate originale:



Mantenendo lo stesso sample rate, tuttavia, l'audio non solo cambierà frequenza, ma anche durata, cosa che si vorrebbe evitare.

3.2 Time stretch con Phase Vocoder

Arrivati a questo punto, vorremmo poter svincolare la durata dell'audio dalla sua intonazione. È possibile farlo tramite un time-stretcher, che consente di modificare la durata dell'audio senza alterarne l'intonazione, sfruttando un approccio basato sul **phase vocoder**: esso lavora nel dominio della frequenza, andando a modificare la fase di ogni singolo spettro ottenuto tramite **Short-Time Fourier Transform** (STFT).

L'idea alla base del time stretching con un phase vocoder è quella di costruire una nuova STFT interpolando opportunamente quella originale (sia in modulo che in fase), e successivamente ricostruire la sequenza nel dominio del tempo mediante la **Inverse Short-Time Fourier Transform** (I-STFT). Tale procedura permette di ottenere una nuova sequenza:

- di lunghezza diversa rispetto all'originale
- ma con stesso pitch

Tutto ciò può essere benissimo gestito sia su MATLAB che su Python utilizzando, rispettivamente, le funzioni built-in o importando la libreria SciPy.

- MATLAB:

```
function x_stretched = time_stretch(x, sr,
stretch_factor)
    % Short-Time Fourier Transform
```

```
4
5       hop_length = 512;
6       nfft = 2048;
7       window = hann(nfft, "periodic");
8
9       [S, ~, T] = stft(x, sr, 'Window', window, ...
10          'OverlapLength', nfft - hop_length, ...
11          'FFTLength', nfft);
12
13      % Calcolo del modulo e della fase
14      magnitude = abs(S);
15      phase = angle(S);
16
17      % Generazione time steps
18      % se stretch_factor > 1, l'audio sara' piu' breve
19      % se stretch_factor < 1, l'audio sara' piu' lungo
20      time_steps = 1 : stretch_factor : size(T, 1);
21
22      % Inizializzazione nuova STFT
23      new_S = zeros(size(S,1), length(time_steps));
24
25      % l'accumulatore svolge il ruolo dell'integrale
26      phase_accumulator = phase(:, 1);
27      og_stft_length = size(S, 2);
28      i = 1;
29      for step = time_steps
30          index = floor(step);
31          fraction = step - index;
32
33          % interpolazione tra i moduli
34          mag1 = magnitude(:, uint64(index));
35          mag2 = magnitude(:, min(uint64(index) + 1,
36          og_stft_length));
37          interpolated_magnitude = (1 - fraction) *
38          mag1 + fraction * mag2;
39
40          % calcolo della frequenza istantanea (derivata)
41          phase1 = phase(:, index);
42          phase2 = phase(:, min(index + 1, og_stft_length)
                );
43          instant_freq = phase2 - phase1;
44
45          % wrapping della frequenza istantanea
46          instant_freq = mod(instant_freq + pi, 2 * pi) -
                pi;
47
48          % aggiornamento dell'accumulatore (integrale)
49          phase_accumulator = phase_accumulator +
                instant_freq;
50
```

```
        new_S(:, i) = interpolated_magnitude .*
        exp(1j * phase_accumulator);

        i = i + 1;
    end
    x_stretched = real(istft(new_S, sr, 'Window', window
        , ...
        'OverlapLength', nfft - hop_length, ...
        'FFTLength', nfft));


    % dissolvenza di volume all'inizio e alla fine:
    L = length(x_stretched(1:end));
    smooth_window = zeros(L, 1);
    dt = 0.1; %sec
    t1 = dt * sr;
    t2 = L - t1;
    smooth_window(1:uint32(t1)) = (0:t1-1)/t1;
    smooth_window(uint32(t1):L-uint32(t1)-1) = 1;
    smooth_window(L-uint32(t1):L) = (L:-1:t2)/t1 - L/t1
        +1;

    x_stretched = smooth_window.*x_stretched;


    % normalizzazione per evitare clipping
    max_value = max(abs(x_stretched));
    if max_value > 0.99
        x_stretched = x_stretched/max_value;
    end
end
```

- Python:

```
import numpy as np
import scipy as sp

def time_stretch(x, fs, stretch_factor, frame_length
    =2048,
hop_length=128):
# Short-Time Fourier Transform
f, t, S = sp.signal.stft(x, fs=fs, nperseg=frame_length,
noverlap=frame_length-hop_length, window=np.hanning(
    frame_length))

# Calcolo del modulo e della fase:
magnitude = np.abs(S)
phase = np.angle(S)

% Generazione time steps
```

```
% se stretch_factor > 1 , l'audio sara' piu' breve
% se stretch_factor < 1 , l'audio sara' piu' lungo
time_steps = np.arange(0, len(t), stretch_factor)

# Inizializzazione nuova STFT
new_S = np.zeros(shape=(np.shape(S)[0], len(time_steps))
    ,
dtype=np.complex64)

# l'accumulatore svolge il ruolo dell'integrale
phase_accumulator = phase[:, 0]

for i, step in enumerate(time_steps):
    index = int(np.floor(step))
    fraction = step - index

    # interpolazione tra i moduli:
    mag1 = magnitude[:, index]
    mag2 = magnitude[:, min(index + 1,
    np.shape(magnitude)[1] - 1)]
    interpolated_magnitude = (1 - fraction) *
    mag1 + fraction * mag2

    # calcolo della frequenza istantanea (derivata):
    phase1 = phase[:, index]
    phase2 = phase[:, min(index + 1,
    np.shape(magnitude)[1] - 1)]
    instant_freq = phase2 - phase1

    # wrapping della frequenza istantanea
    instant_freq = np.mod(instant_freq + np.pi,
    2 * np.pi) - np.pi

    # aggiornamento dell'accumulatore (integrale)
    phase_accumulator += instant_freq #stretch_factor

    new_S[:, i] = interpolated_magnitude
    * np.exp(1j * phase_accumulator)

_, x_stretched = sp.signal.istft(new_S, fs=fs, nperseg=
    frame_length,
noverlap=frame_length-hop_length)
return x_stretched
```

3.3 Pitch shift

Dato che l'obiettivo principale è quello di modificare solo e unicamente la frequenza, è possibile un *pitch shift* attraverso una combinazione di **time stretch** e **resampling** per raggiungere tale obiettivo. Qualora volessimo moltiplicare

la frequenza f_o di un fattore pari a k, posto L pari alla durata della traccia, la procedura da seguire è la seguente:

- **time stretch** di un fattore $\frac{1}{k} \Rightarrow$ durata $= \frac{L}{k}$; frequenza $= f_o$
- **resample** con step pari a $k \Rightarrow$ durata $= L\frac{k}{k} = L$; frequenza $= kf_o$

Al termine di ciò, la durata del nuovo audio coinciderà con quella dell'audio originale - o quasi, ci potrebbero essere degli errori di approssimazione dovuti al calcolo di $\frac{1}{k}$, dunque sarà necessario ricampionare un ulteriore volta per mantenere lo stesso numero di campioni presenti inizialmente - ma l'intonazione sarà del tutto nuova. A livello implementativo, tale tecnica può essere realizzata in poche righe di codice:

- MATLAB:

```
function x_pitched = pitch_shifting(x, sr, pitch_factor)
    x_stretched = time_stretch(x, sr, 1/pitch_factor);

    % viene fatto uso della funzione resample
    % precedentemente implementata:
    x_pitched = resample(x_stretched, sr, pitch_factor);

    % la lunghezza del segnale in ingresso x deve essere
    % uguale a quella del segnale in uscita x_pitched:
    old_t = 1:length(x_pitched);
    new_t = 1:length(x);
    x_pitched = interp1(old_t, x_pitched, new_t);
end
```

- Python:

```
def pitch_shifting(x, fs, pitch_factor,
frame_length=2048, hop_length=512):
    x_stretched = time_stretch(x, fs, 1/pitch_factor,
    frame_length, hop_length)
    stretched_length = len(x_stretched)

    t_old = np.arange(0, stretched_length)
    t_new = np.arange(0, stretched_length, pitch_factor)
    x_pitched = np.interp(t_new, t_old, x_stretched)

    t_old = np.arange(0, len(x_pitched))
    t_new = np.arange(0, len(x))
    x_pitched = np.interp(t_new, t_old, x_pitched)

    return x_pitched
```

4 Pitch detection

4.1 Utilizzo dell'autocorrelazione

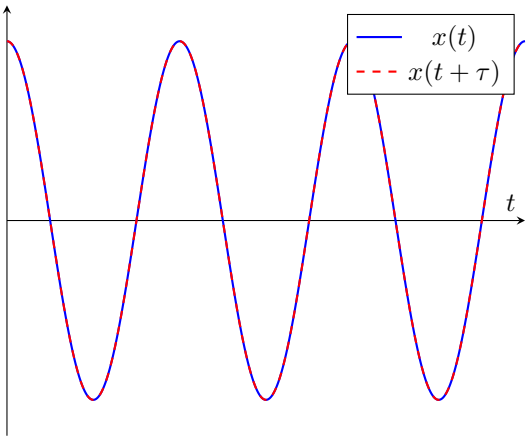
A questo punto, è necessario sviluppare un algoritmo in grado di identificare con precisione l'intonazione di una linea vocale, al fine di correggere e adattare la nota desiderata. Ciò potrebbe essere implementato nel dominio del tempo attraverso l'autocorrelazione. Noto che la frequenza fondamentale f_0 è il reciproco del periodo T, ci basterà calcolare il periodo per risalire alla frequenza fondamentale.

Per introdurre il concetto, consideriamo un segnale di test costituito da un coseno puro:

$$x(t) = \cos(t)$$

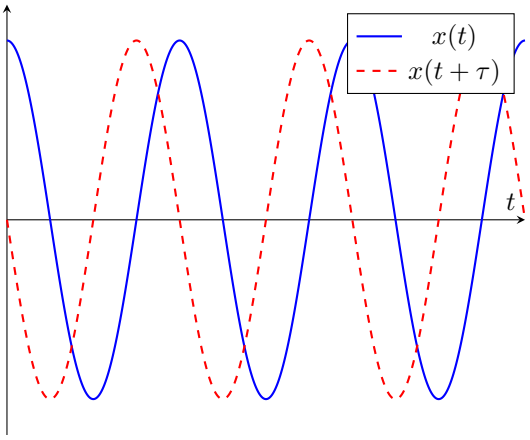
e calcoliamone l'autocorrelazione per $\tau \in \{0, \frac{\pi}{2}, \pi, \frac{3}{2}\pi, 2\pi\}$. Per semplicità, eseguiremo l'integrale su un intervallo finito.

$\tau = 0$



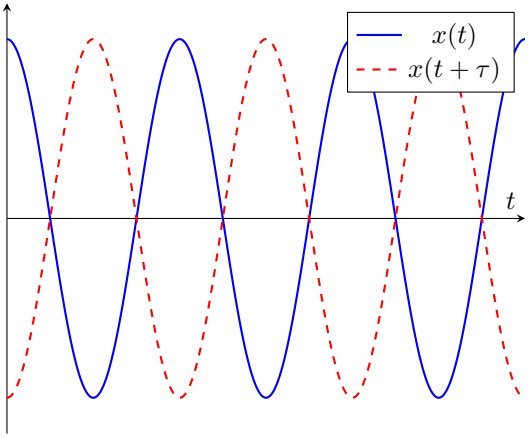
$$R_{xx}^{(E)}(0) = \int_0^{2\pi} x(t)\, x(t+0)\ dt = \int_0^{2\pi} x^2(t)\ dt = \pi \simeq 3.1416 \tag{8}$$

$\tau = \frac{\pi}{2}$



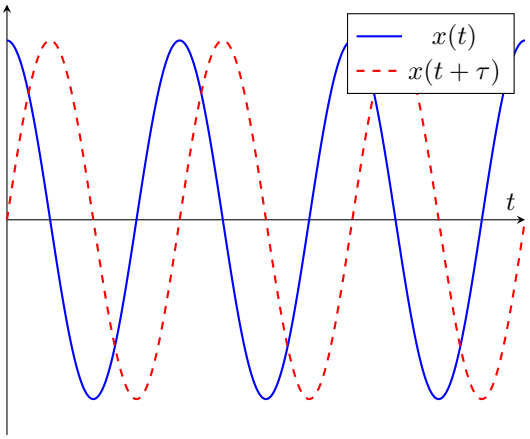
$$R_{xx}^{(E)}\left(\frac{\pi}{2}\right) = \int_0^{2\pi} x(t)\, x\left(t+\frac{\pi}{2}\right)\, dt = 0 \tag{9}$$

$\tau = \pi$



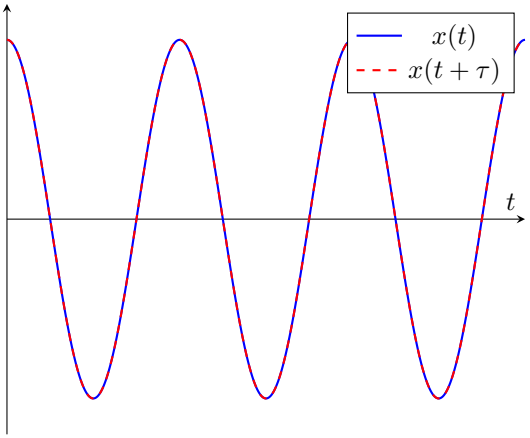
$$R_{xx}^{(E)}(\pi) = \int_0^{2\pi} x(t)\, x(t+\pi)\, dt = -\pi \simeq -3.1416 \tag{10}$$

$\tau = \frac{3}{2}\pi$



$$R_{xx}^{(E)}\left(\frac{3}{2}\pi\right) = \int_0^{2\pi} x(t)\, x\left(t + \frac{3}{2}\pi\right)\, dt = 0 \tag{11}$$

$\tau = 2\pi$



$$R_{xx}^{(E)}\left(2\pi\right) = \int_0^{2\pi} x(t)\, x\left(t + 2\pi\right)\, dt = \pi \simeq 3.1416 \tag{12}$$

Alla luce di ciò, quando τ coincide con il periodo, l'autocorrelazione raggiunge lo stesso valore dell'origine, generando un massimo. Il primo massimo relativo della linea melodica permette quindi di determinare la frequenza fondamentale.

Nella pratica, però, le cose sono diverse. Innanzitutto, lavoriamo con segnali numerici, di lunghezza finita e discretizzati, per cui il calcolo si riduce a una

sommatoria. Inoltre, i segnali reali non sono perfette sinusoidi: per questo motivo, il metodo teorico visto in precedenza non risulta applicabile in modo diretto.

4.2 Squared Difference Function

Per ottenere una maggiore precisione nel calcolo, invece di valutare l'area del prodotto di due funzioni traslate, risulta più efficace calcolare l'area della loro differenza al quadrato. Questa funzione prende il nome di *Funzione di Differenza al Quadrato* o *Squared Difference Function* (**SDF**). Quando i due segnali si sovrappongono, la loro differenza è minima. La SDF si definisce come:

$$d(\tau) = \frac{1}{N} \sum_{n=0}^{N-1} (x[n] - x[n+\tau])^2 \tag{13}$$

dove:

- N è la lunghezza della finestra considerata;
- il fattore di normalizzazione $\frac{1}{N}$ permette di compensare la riduzione della lunghezza utile della finestra all'aumentare di τ, evitando che la funzione sia influenzata da questo effetto.

Una volta calcolata la SDF, sarà sufficiente individuare il primo minimo locale per determinare il valore di τ corrispondente alla migliore sovrapposizione tra i due segnali. Infine, dopo aver trovato τ, è necessario ricavare il periodo T noto che:

- $T = \frac{1}{f_0}$
- $f_0 = \frac{\tau}{sr}$, con sr pari alla frequenza di campionamento o **sample rate**

dunque:

$$T = \frac{sr}{\tau} \tag{14}$$

Ciò può essere implementato in questo modo:

- MATLAB:

```
function f0 = find_pitch(sample, fs)
    % Intervallo di frequenze vocali (Hz)
    voice_range = [80, 200];
    min_lag = floor(fs / voice_range(2));
    max_lag = floor(fs / voice_range(1));

    % Calcolo delle differenze quadrate normalizzate
    diff_areas = zeros(1, max_lag - min_lag + 1);
    for tau = min_lag:max_lag
        diff_areas(tau - min_lag + 1) = ...
```

```
        get_difference_area(sample, sample(tau+1:end));
    end

    % Ricerca del primo minimo relativo
    delta_lag = 0;
    for tau = 2:length(diff_areas)-1
        if diff_areas(tau) < diff_areas(tau-1) ...
        && diff_areas(tau) < diff_areas(tau+1)
            delta_lag = tau;
            break;
        end
    end

    % Se non trovato, prendo il minimo globale
    if delta_lag == 0
        [~, delta_lag] = min(diff_areas);
    end

    % Conversione in frequenza fondamentale
    f0 = fs / (delta_lag + min_lag - 1);
end


function diff = get_difference_area(x1, x2)
    L = min(length(x1), length(x2));
    diff = sum((x1(1:L) - x2(1:L))^2) / L;
end
```

- Python:

```
def find_pitch(sample, fs):
    voice_range = [80, 200] # Hz
    min_lag = fs // voice_range[1]
    max_lag = fs // voice_range[0]

    diff_areas = [get_difference_area(sample,
    sample[tau:]) for tau in range(min_lag, max_lag)]
    mean = np.sum

    # il primo minimo relativo trovato sara'
    # il tau ricercato
    delta_lag = 0
    for tau in range(1, len(diff_areas) - 1):
        if diff_areas[tau] < diff_areas[tau-1]
        and diff_areas[tau] < diff_areas[tau+1]:
            delta_lag = tau

    if delta_lag == 0:
        delta_lag = np.argmin(diff_areas)

```

```
    return fs / (delta_lag + min_lag)


def get_difference_area(x1, x2):
    length = min(len(x1), len(x2))
    return np.sum((x1[:length] - x2[:length])
    ** 2) / length
```

5 Algoritmo di Tuning Completo

L'algoritmo finale di **tuning** implementa in maniera completa il processo di correzione dell'intonazione di un segnale audio discreto $x[n]$ campionato a frequenza f_s. Esso combina i metodi introdotti nelle sezioni precedenti: segmentazione in finestre, stima della frequenza fondamentale, ricerca della nota da raggiungere, pitch shifting e ricostruzione mediante overlap-add.

5.1 Segmentazione e Finestratura

Il segnale viene suddiviso in frame di lunghezza $L = 2048$ campioni, con hop size pari a $H = L/4$. Ogni frame è moltiplicato per una finestra di Hann $\omega[n]$ al fine di ridurre effetti di discontinuità.

5.2 Stima e Correzione del Pitch

Per ciascun frame viene stimata la frequenza fondamentale $\hat{f}_0$ mediante la funzione già precedentemente implementata **find_pitch**. La nota più vicina appartenente alla scala scelta S viene individuata come

$$f^* = \textbf{find\_closest\_note}(\hat{f}_0, S).$$

Il fattore di correzione è quindi

$$\alpha = \frac{f^*}{\hat{f}_0}.$$

5.3 Pitch Shifting

Il frame viene trasformato tramite la funzione di pitch shifting che è stata realizzata, secondo la relazione:

$$y_i[n] = \texttt{pitch\_shift}(x_i[n], f_s, \alpha),$$

5.4 Overlap-Add con Allineamento

I frame processati vengono ricombinati nel segnale di output $x_{out}[n]$ con tecnica di overlap-add. Per garantire continuità temporale, viene stimato un ritardo τ

che massimizza la correlazione tra frame adiacenti:

$$x_{out}[n] = x_{out}[n] + y_i[n - \tau].$$

In questo modo:

- MATLAB:

```
function best_lag = find_best_lag(current_sample, frame,
    start_index, n_overlap)
    % Calcola il lag massimo consentito
    max_lag = floor(n_overlap / 8);

    % Definisce la regione di ricerca
    search_start = max(1, start_index - max_lag);
    search_end = min(length(current_sample), start_index
        + length(frame) + max_lag);

    search_region = current_sample(search_start:
        search_end);

    % Se la regione    troppo corta, ritorna 0
    if length(search_region) < length(frame)
        best_lag = 0;
        return;
    end

    % Calcola la correlazione incrociata
    correlation = xcorr(search_region, frame, 'none');

    % Trova   l offset   migliore
    [~, best_offset] = max(correlation);

    % Calcola il lag relativo alla posizione originale
    best_lag = (search_start + best_offset - 1) -
        start_index;
end
```

- Python:

- MATLAB:

```
def find_best_lag_2(current_sample, frame, start_index,
    n_overlap):
    max_lag = n_overlap // 8

    # Estrai la regione di interesse dal segnale
        corrente
    search_start = max(0, start_index - max_lag)
    search_end = min(len(current_sample), start_index +
        len(frame) + max_lag)
```

```
    search_region = current_sample[search_start:
        search_end]

    if len(search_region) < len(frame):
        return 0

    # Usa correlazione crociata
    correlation = np.correlate(search_region, frame,
        mode='valid')
    best_offset = np.argmax(correlation)

    # Converti in lag relativo alla posizione originale
    best_lag = search_start + best_offset - start_index

    return best_lag
```

5.5 Normalizzazione Finale

Infine, il segnale viene normalizzato per evitare fenomeni di clipping:

$$x_{out} \leftarrow \frac{x_{out}}{\max(x_{out})}.$$

5.6 Output

La funzione restituisce x_{out}, un segnale in cui ogni frame è stato intonato alla nota più vicina della scala musicale selezionata, con transizioni continue e senza artefatti percettibili.

5.7 Implementazione

- MATLAB:

```
function x_tuned = tuning(data, sr, scale, note_dict, A)
    sample_length = numel(data);
    frame_length = 2048;
    hop_length = frame_length / 4;

    x_tuned = zeros(size(data));

    % overlap add
    window = hamming(frame_length);

    for ii = (1: hop_length: sample_length)
        if ii + frame_length >= sample_length
            frame = zeros(frame_length, 1);
            frame(1:sample_length - (ii-1)) = data(ii:
                sample_length);
        else
```

```
            frame = data(ii : ii + frame_length-1);
        end


        % multiplying frame by window function
        frame = frame.* window;


        % finding frame pitch
        pitch = find_pitch(frame, sr);

        goal_pitch = find_closest_note(pitch, note_dict(
            scale), note_dict, A);

        %shift_factor = goal_pitch/pitch;
        shift_factor = 0.9;
        frame = pitch_shifting(frame, sr, shift_factor);
        tau = 0;
        if ii ~= 1
            tau = find_best_lag(x_tuned, frame, ii,
                frame_length - hop_length);
        end

        % add

        if ii + tau + frame_length >= sample_length
            x_tuned(ii + tau : end) = x_tuned(ii + tau :
                 end) + frame(1: sample_length - ii -
                tau + 1);
        else
            temp = x_tuned(ii + tau : ii + tau +
                frame_length-1) + frame;
            x_tuned(ii + tau : ii + tau + frame_length
                -1) = temp;
        end
    end

    % Normalize output
    max_value = max(x_tuned);
    clip = 1;
    if max_value > clip
        x_tuned = x_tuned / (max_value/clip);
    end
end
```

- Python:

```
def tuning(x, fs, scale="chromatic"):
    sample_length = len(x)

```

```
    frame_length = int(2048)
    hop_length = frame_length // 4

    x_out = np.zeros_like(x)

    # Overlap-add
    window = np.hanning(frame_length)
    for i in range(0, sample_length, hop_length):
        if i + frame_length >= sample_length:
            frame = np.zeros(shape=(frame_length,))
            frame[: sample_length - i] = x[i :
                sample_length]
        else:
            frame = x[i : i + frame_length]

        # multiplying frame by window function
        frame = frame * window

        # finding frame pitch
        pitch = find_pitch(frame, fs)

        goal_pitch = find_closest_note(pitch, dict_scale
            [scale])


        # shifting to the aim pitch
        frame = pitch_shifting(frame, fs, goal_pitch/
            pitch)

        tau = find_best_lag(x_out, frame, i,
            frame_length - hop_length)

        # Add
        if i + tau + frame_length >= sample_length:
            x_out[i + tau : sample_length] = x_out[i +
                tau : sample_length] + frame[:
                sample_length - i - tau]
        else:
            x_out[i + tau : i + tau + frame_length] =
                x_out[i + tau: i + tau + frame_length] +
                 frame


    # Normalize output
    max_value = np.max(x_out)
    clip = 1
    if max_value > clip:
        x_out /= (max_value/clip)

    return x_out
```

6 Conclusioni e sviluppi futuri

L'implementazione descritta costituisce una versione prototipale di un algoritmo di *autotuning*. Pur dimostrando l'efficacia concettuale della tecnica di segmentazione, rilevamento della frequenza fondamentale e ricampionamento con overlap-add, essa rimane inevitabilmente una versione semplificata rispetto a soluzioni di livello professionale.

In particolare, si evidenziano alcune limitazioni che potranno costituire linee guida per sviluppi futuri:

- **Resampling sinc-based**: l'attuale approccio al cambio di pitch si affida a trasformazioni di libreria che non garantiscono un'ottimale qualità del segnale. Un ricampionamento sinc-based permetterebbe di preservare maggiormente la fedeltà spettrale.
- **Mantenimento delle formanti**: l'algoritmo non si occupa della conservazione delle formanti vocali, con il rischio di introdurre artefatti timbrici percepibili, soprattutto su segnali vocali complessi.
- **Efficienza computazionale**: l'implementazione è stata sviluppata in linguaggi interpretati (MATLAB e Python), con l'uso estensivo di cicli `for`. Ciò comporta tempi di esecuzione non ottimali, che potrebbero essere sensibilmente ridotti mediante l'uso di tecniche di vectorization o con una successiva traduzione in linguaggi compilati.

In sintesi, il lavoro fornisce una base solida e didatticamente chiara per comprendere i meccanismi fondamentali del pitch shifting automatico, pur lasciando ampi margini di miglioramento sia dal punto di vista della qualità percettiva del segnale che dell'efficienza implementativa.