Metodi Numerici Avanzati

Riduzione del rumore in un segnale audio tramite SVD

Docente:
Elisa Francomano

Autori:
Gabriele Baiamonte
Riccardo Rubino

Anno Accademico 2024/2025

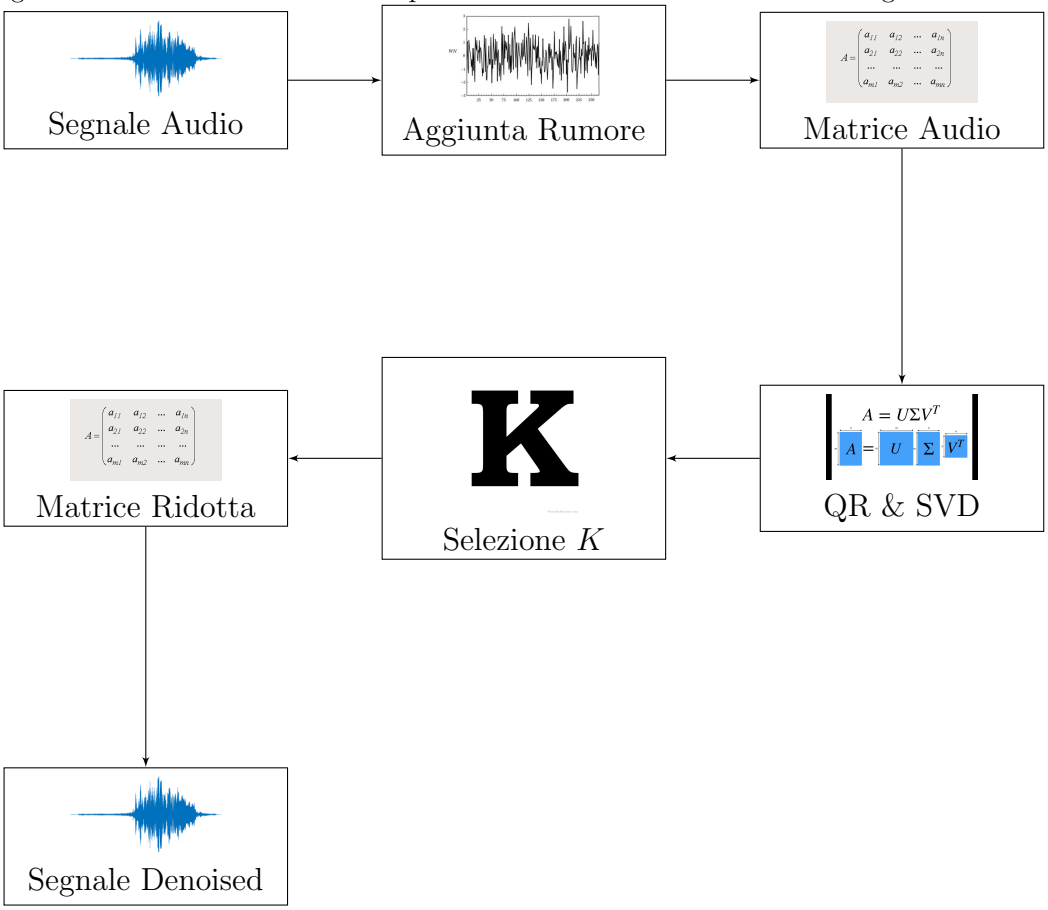
Indice

1 Introduzione

La riduzione del rumore nei segnali audio trova molte applicazioni concrete, viene implementata in tutte le tecnologie che permettono lo scambio di segnali audio come la telefonia e il videochatting, ma trova anche applicazioni nell'intelligenza artificiale in particolare nell'ambito della sintesi vocale. L'implementazione di questa tecnologia si basa su alcuni metodi matematici come la SVD. Vedremo in dettaglio l'idea progettuale alla base di questa tecnologia, i fondamenti teorici che riguardano i metodi matematici utilizzati con la loro implementazione e nell'ultima parte analizzeremo i risultati.

2 Schema a blocchi dell'algoritmo

Il seguente schema a blocchi mostra il processo di riduzione del rumore in un segnale audio:



Ad un segnale audio di partenza viene aggiunto un rumore di tipo Gaussiano e, successivamente, il segnale viene convertito in una matrice audio alla quale abbiamo applicato la decomposizione ai valori singolari realizzata tramite il metodo QR per il calcolo delle componenti delle matrici della SVD.

Successivamente si effettua una selezione del parametro K, che rappresenta il numero di valori singolari da preservare. La scelta del parametro K risulta essere cruciale poiché un K troppo basso rimuoverebbe una quantità eccessiva di informazioni, portando a una qualità audio molto bassa eliminando gran parte del rumore ma anche componenti del segnale importanti, mentre un K troppo elevato preserverebbe troppo del segnale originale, riducendo l'efficacia nella rimozione del rumore; dunque è necessario effettuare una scelta accurata del parametro K tale da mantenere un buon compromesso.

Successivamente alla scelta del parametro si ottiene la matrice ridotta tramite SVD che viene riconvertita in un segnale audio.

3 Fondamenti teorici

3.1 SVD

3.1.1 Cos'è l'SVD

La decomposizione ai valori singolari, anche detta SVD, è una tecnica di fattorizzazione di una matrice A reale o complessa di dimensione mxn basata sull'utilizzo di autovalori ed autovettori.

Teorema di esistenza della SVD: Data una qualunque matrice $A \in R^{mxn}$, esistono $U \in R^{mxm}$, $V \in R^{nxn}$ ortogonali e $\Sigma \in R^{mxn}$ tali che:

$$A = U\Sigma V^T$$

1. U è composta dai vettori u_i che sono gli autovettori di AA^T detti vettori singolari sinistri
2. V è composta dai vettori v_i che sono gli autovettori di A^TA detti vettori singolari destri.

Invece la matrice Σ è una matrice pseudo-diagonale di dimensione mxn composta come segue:

$$\Sigma_{ij} = \begin{cases} 0 & \text{se } i \neq j, \\ \sigma_i & \text{se } i = j. \end{cases} \tag{1}$$

Gli elementi σ_i che compongono la matrice Σ sono detti valori singolari di A e $\sigma_i = \sqrt{\lambda_i}$ dove i λ_i sono gli autovalori della matrice A^TA.
In particolare i valori singolari sono disposti nella matrice Σ in ordine decrescente:

$$\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq ... \geq \sigma_p \geq 0$$
$$p = min(m, n)$$

Dimostrazione:
Per dimostrare l'esistenza della SVD si procede per induzione sulla dimensione di A. Dati i vettori $\mathbf{v}_1 \in \mathbb{R}^n$ e $\mathbf{u}_1 \in \mathbb{R}^m$ tali che $\|\mathbf{v}_1\|_2 = \|\mathbf{u}_1\|_2 = 1$ e $A\mathbf{v}_1 = \sigma_1 \mathbf{u}_1$,

$$\sigma_1 = \|A\|_2 \quad \text{dato che } \|A\|_2 = \sqrt{\rho(A^T A)}.$$

Consideriamo ora le matrici $\tilde{V}_1 \in \mathbb{R}^{n\times(n-1)}$ e $\tilde{U}_1 \in \mathbb{R}^{m\times(m-1)}$ tali che

$$V_1 = [\mathbf{v}_1, \tilde{V}_1] \in \mathbb{R}^{n\times n} \quad \text{e} \quad U_1 = [\mathbf{u}_1, \tilde{U}_1] \in \mathbb{R}^{m\times m}$$

risultino essere ortogonali.
Poiché

$$\mathbf{u}_1^T A\mathbf{v}_1 = \mathbf{u}_1^T \sigma_1 \mathbf{u}_1 = \sigma_1 \mathbf{u}_1^T \mathbf{u}_1 = \sigma_1 \|\mathbf{u}_1\|_2 = \sigma_1$$

e

$$\tilde{U}_1^T A\mathbf{v}_1 = \tilde{U}_1^T \sigma_1 \mathbf{u}_1 = \sigma_1 \tilde{U}_1^T \mathbf{u}_1 = 0$$

(dato che il vettore $\mathbf{u}_1$ è ortogonale ad ogni colonna di $\tilde{U}_1$)

possiamo quindi scrivere:

$$U_1^T A V_1 = \begin{bmatrix} \mathbf{u}_1 \\ \tilde{U}_1 \end{bmatrix} A \begin{bmatrix} \mathbf{v}_1 & \tilde{V}_1 \end{bmatrix} = \begin{bmatrix} \sigma_1 & w^T \\ 0 & B \end{bmatrix} = A_1,$$

con $w \in \mathbb{R}^{n-1}$ e $B \in \mathbb{R}^{(m-1)\times(n-1)}$.
Ora:

$$\begin{aligned} \left\| A_1 \begin{bmatrix} \sigma_1 \\ w \end{bmatrix} \right\|_2^2 &= \left\| \begin{bmatrix} \sigma_1 & w^T \\ 0 & B \end{bmatrix} \begin{bmatrix} \sigma_1 \\ w \end{bmatrix} \right\|_2^2 = \left\| \begin{bmatrix} \sigma_1^2 + w^T w \\ Bw \end{bmatrix} \right\|_2^2 \\ &= (\sigma_1^2 + w^T w)^2 + \|Bw\|_2^2 \\ &\geq (\sigma_1^2 + w^T w)^2. \end{aligned}$$

E quindi:

$$\|A_1\|_2^2 \geq (\sigma_1^2 + w^T w).$$

Ma $\|A\|_2^2 = \sigma_1^2$ e $\|A\|_2 = \|A_1\|_2$ dato che A_1 si ottiene tramite trasformazioni ortogonali di A, da cui:

$$\|A_1\|_2^2 = \|A\|_2^2 = \sigma_1^2 \geq (\sigma_1^2 + w^T w) = (\sigma_1^2 + \|w\|_2^2).$$

Quindi:

$$0 \geq \|w\|_2^2 \quad \Rightarrow \quad \|w\|_2^2 = 0.$$

Allora:

$$A_1 = \begin{bmatrix} \sigma_1 & 0 \\ 0 & B \end{bmatrix}.$$

Notiamo che abbiamo dimostrato il caso base dell'induzione dato che per $n = 1$ o $m = 1$ abbiamo ottenuto la decomposizione SVD. Dunque supponiamo che il teorema sia vero per una matrice $((m-1)\text{x}(n-1))$. Dimostriamo ora che è vero per una matrice generica ($A \in \mathbb{R}^{mxn}$). Per ipotesi induttiva B avrà una decomposizione SVD del tipo $B = U_2 \Sigma_2 V_2^T$, da cui:

$$A = U_1 A_1 V_1^T = U_1 \begin{bmatrix} \sigma_1 & 0 \\ 0 & B \end{bmatrix} V_1^T = U_1 \begin{bmatrix} \sigma_1 & 0 \\ 0 & U_2 \Sigma_2 V_2^T \end{bmatrix} V_1^T =$$

$$= U_1 \begin{bmatrix} 1 & 0 \\ 0 & U_2 \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & V_2^T \end{bmatrix} V_1^T$$

Si verifica facilmente che questa è una decomposizione SVD di A dato che:

- $U = U_1 \begin{bmatrix} 1 & 0 \\ 0 & U_2 \end{bmatrix}$ è una matrice ortogonale in quanto prodotto di due matrici ortogonali e lo stesso ragionamento vale per V.
- $\Sigma = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix}$ è per costruzione una matrice diagonale contenente i valori singolari di A.

3.1.2 Come sono composte le matrici V, Σ e U?

Consideriamo il prodotto $A^T A$:

$$A^T A = (V\Sigma^T U^T)(U\Sigma V^T) = V\Sigma^T \Sigma V^T$$

Questo può essere riscritto come:

$$(A^T A)V = V(\Sigma^T \Sigma)$$

Ciò implica che la matrice $\Sigma^T\Sigma$ è la matrice degli autovalori di $A^T A$ e V è la matrice degli autovettori di $A^T A$. In particolare, si osserva che $A^T A$ è una matrice semidefinita positiva e simmetrica, il che significa che i suoi autovettori, disposti come colonne, formano una matrice ortogonale V e i suoi autovalori sono positivi.

La matrice $\Sigma^T\Sigma$ ha dimensione $\max(n,m) \times \max(n,m)$ e contiene sulla diagonale i quadrati dei valori singolari, infatti banalmente nel caso in cui Σ abbia dimensione 4×6.

$$\mathbf{\Sigma} = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_4 & 0 & 0 \end{bmatrix} \quad \mathbf{\Sigma^T} = \begin{bmatrix} \sigma_1 & 0 & 0 & 0 \\ 0 & \sigma_2 & 0 & 0 \\ 0 & 0 & \sigma_3 & 0 \\ 0 & 0 & 0 & \sigma_4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{D = \Sigma^T \Sigma} = \begin{bmatrix} \sigma_1^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma_2^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_3^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma_4^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$(A^T A)V = V(\Sigma^T \Sigma) = VD$$

Possiamo quindi affermare che gli elementi di D corrispondono agli autovalori di $A^T A$:

$$\lambda_i(A^T A) = \sigma_i^2 \quad \Longrightarrow \quad \sigma_i = \sqrt{\lambda_i(A^T A)}$$

Analogalmente, consideriamo il prodotto AA^T:

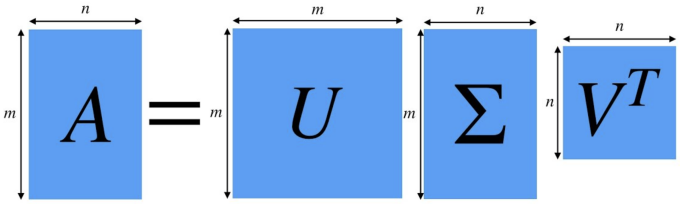
$$AA^T = (U\Sigma V^T)(V\Sigma^T U^T) = U\Sigma\Sigma^T U^T$$

Da cui:

$$(AA^T)U = U(\Sigma\Sigma^T)$$

Ovvero, U è la matrice degli autovettori di AA^T.

3.1.3 Approssimazione della matrice A



Posto $p = min(m, n)$ possiamo scrivere la decomposizione come:

$$\mathbf{A} = \sum_{i=1}^{p} \mathbf{u}_i \sigma_i \mathbf{v}_i^{\mathrm{T}}$$

Da cui si può intuire come i primi elementi della sommatoria, ovvero le prime matrici che compongono A, contengono informazioni maggiori rispetto a quelle più "in fondo" alla sommatoria dato che i pesi σ_i vanno decrescendo.

Teorema Se per qualche $1 \leq r \leq p$ risulta che:

$$\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_r > \sigma_{r+1} = ... = \sigma_p = 0$$

Ovvero da un certo r in poi i valori singolari risultano essere nulli, allora:

- $rank(A) = r$
- $\mathbf{A} = \sum_{i=1}^{r} \mathbf{u}_i \sigma_i \mathbf{v}_i^{\mathrm{T}}$ (**decomposizione spettrale**)

Questo significa che le dimensioni dalla $r + 1esima$ alla $p - esima$ della matrice A sono combinazioni lineari delle prime r.

Approssimazione lower rank Data $A \in R^{mxn}$ con $rank(A) = r$, possiamo stabilire un $k < r$ e formulare un'approssimazione A_k della matrice A:

$$\mathbf{A_k} = \sum_{i=1}^{k} \mathbf{u}_i \sigma_i \mathbf{v}_i^{\mathrm{T}}$$

L'errore che si commette adottando tale approssimazione:

$$\beta = \{B \in R^{mxn} | rank(A) = k\}$$

$$\min_{B \in \beta} \|A - B\|_2 = \|A - A_k\|_2 = \left\| \sum_{i=k+1}^{r} \mathbf{u}_i \sigma_i \mathbf{v}_i^{\mathrm{T}} \right\|_2 = \left\| \tilde{U} \tilde{\Sigma} \tilde{V}^T \right\|_2 = \sigma_{k+1}$$

Quindi l'approssimazione migliora al crescere di k.

3.2 Fattorizzazione QR

3.2.1 Cos'è la Fattorizzazione QR

La fattorizzazione QR permette di fattorizzare una matrice A di dimensioni nxn nel prodotto di due matrici Q e R.

$$A = QR$$

dove abbiamo:

- Una matrice ortogonale Q.

> **Definizione:** Una matrice Q è detta ortogonale se soddisfa la seguente condizione:
> $$Q^T Q = QQ^T = I$$
> dove Q^T è la trasposta di Q e I è la matrice identità.

- Una matrice triangolare superiore R.

> **Definizione:** Una matrice R di ordine $n \times n$ è detta triangolare superiore se tutti gli elementi al di sotto della diagonale principale sono uguali a zero. In altre parole, la matrice R ha la forma:
> $$R = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ 0 & 0 & \cdots & r_{n-1,n} \\ 0 & 0 & \cdots & r_{nn} \end{pmatrix}$$

quindi nel dettaglio otteniamo:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1n} \\ q_{21} & q_{22} & \cdots & q_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ q_{n1} & q_{n2} & \cdots & q_{nn} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{bmatrix}.$$

Per effettuare la fattorizzazione QR vedremo 3 metodi:

- Algoritmo di Gram-Schmidt.
- Matrici di trasformazione Householder.
- Matrici di trasformazione Givens.

3.2.2 Algoritmo di Gram-Schmidt

L'algoritmo di Gram-Schmidt permette di costruire i vettori ortogonali della matrice Q partendo dalla matrice A.
Data la nostra matrice A:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

il j-esimo vettore colonna q della matrice Q viene costruito rendendolo ortogonale alle colonne precedenti $(q^{(1)}, q^{(2)}, \ldots)$.
Per calcolare $q^{(1)}$ normalizziamo la prima colonna della matrice A:

$$q^{(1)} = \frac{a^{(1)}}{\|a^{(1)}\|}$$

Ogni colonna successiva $q^{(j)}$ viene calcolata con la seguente formula:

$$\hat{q}^{(j)} = a^{(j)} - \sum_{i=1}^{j-1} (q^{(i)} \cdot a^{(j)}) q^{(i)}$$

dove $(q^{(i)} \cdot a^{(j)})q^{(i)}$ è la proiezione di $a^{(j)}$ su $q^{(i)}$.

Definizione: Proiettare un vettore: Dato un vettore $a^{(j)}$ e un vettore $q^{(i)}$, la proiezione di $a^{(j)}$ su $q^{(i)}$ è definita come

$$(q^{(i)} \cdot a^{(j)})q^{(i)}$$

Dove $(q^{(i)} \cdot a^{(j)})$ è il prodotto scalare tra $q^{(i)}$ e $a^{(j)}$, che misura quanto $a^{(j)}$ punta nella direzione di $q^{(i)}$, mentre moltiplicando per $q^{(i)}$ si ottiene un vettore nella stessa direzione di $q^{(i)}$ ma scalato.

Una volta ottenuto il vettore ortogonale $\hat{q}^{(j)}$ lo normalizziamo ottenendo:

$$q^{(j)} = \frac{\hat{q}^{(j)}}{\|\hat{q}^{(j)}\|}$$

I valori presenti nella matrice R sono i coefficienti delle proiezioni, formalmente:

$$r_{ij} = q^{(i)} \cdot a^{(j)} \quad \text{per } i \leq j$$

Implementazione dell'algoritmo di Gram-Schmidt Vediamo un'implementazione dell'algoritmo trattato precedentemente

```
function [Q, R] = Gram_Schmidt(A)

    [M, N] = size(A);
    v1 = A(:, 1);
    R(1, 1) = norm(v1);
    Q(:, 1) = v1 / R(1, 1); % Normalizza la prima colonna e la
        inserisce in Q

    for n = 2:min(M, N) % Itera su ciascuna colonna
        vn = A(:, n);
        R(1:n-1, n) = Q(:, 1:n-1)' * vn; % Proietta vn sulle
            colonne gia' calcolate di Q
        qn = vn - Q(:, 1:n-1) * R(1:n-1, n); % Sottrae la
            componente proiettata
        R(n, n) = norm(qn);
        Q(:, n) = qn / R(n, n); % Normalizza e lo inserisce come
            colonna di Q
    end
end
```

Costo computazionale e stabilità numerica Analizziamo il costo computazionale dell'algoritmo: Dati:

1. $A \in \mathbb{R}^{M \times N}$, con M righe e N colonne.

Il metodo di Gram-Schmidt calcola una base ortogonale tramite le seguenti operazioni per ogni colonna n:

1. **Proiezione:** Proiettare il vettore $\mathbf{v}_n$ sulle colonne già ortogonalizzate di Q, con costo:

$$\mathcal{O}((n-1)M)$$

2. **Normalizzazione:** Calcolare la norma e normalizzare il vettore, con costo:

$$\mathcal{O}(M)$$

Il costo per ciascuna colonna n è quindi:

$$\mathcal{O}(nM).$$

dato che abbiamo N colonne il costo

$$\mathcal{O}(MN^2).$$

Per quanto riguarda la stabilita numerica, l'algoritmo risulta instabile per via delle sottrazioni presenti nel calcolo delle proiezioni che possono portare al fenomeno della cancellazione numerica.

3.2.3 Matrici di trasformazione Householder

Uno strumento utilizzato per effettuare la fattorizzazione QR sono le matrici di Householder, definiamo dapprima come è formata una matrice di Householder:

Definizione:

$$H = I - 2ww^T$$

dove

$$\|w\|_2 = 1$$

Le proprietà fondamentali di questa matrice sono le seguenti:

- H è una matrice simmetrica:

$$H^T = (I - 2ww^T)^T = I - 2ww^T = H$$

- H è ortogonale

$$\begin{aligned} H^T H &= (I - 2ww^T)^T (I - 2ww^T) = (I - 2ww^T)(I - 2ww^T) = \\ &= I - 2ww^T - 2ww^T + 4ww^T ww^T = I - 4ww^T + 4ww^T = I. \end{aligned}$$

Da queste proprieta si dimostra banalmente che H è anche involutoria

$$H = H^{-1}$$

Dimostrazione:

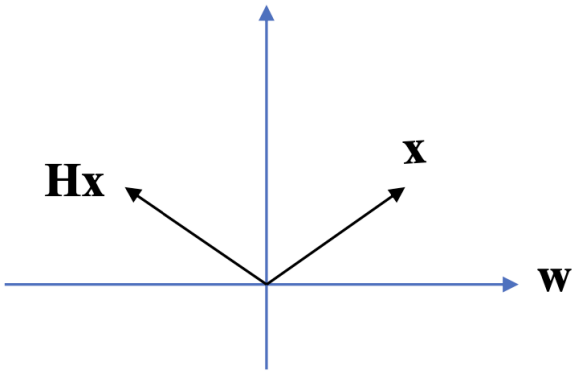
Per definizione di matrice ortogonale, abbiamo:

$$H^T H = I.$$

Moltiplicando ambo i membri a sinistra per H^{-1}, otteniamo:

$$\begin{aligned} H^{-1} H^T H &= H^{-1} I \\ I H^T &= H^{-1} \\ H^T &= H^{-1}. \end{aligned}$$

La matrice di Householder è definita anche matrice di riflessione, essa permette di riflettere un vettore $x \neq 0$ rispetto a una direzione ortogonale a un vettore w. Con riflessione rispetto a w si intende che il vettore Hx si trova in posizione opposta rispetto a x ma rimanendo simmetrico alla direzione ortogonale a w.



Dopo aver compreso le proprietà fondamentali delle matrici di Householder e il suo significato geometrico, analizzeremo un concetto che ci tornerà utile nella fattorizzazione QR, ossia come le matrici di Householder ci consentono di trasformare un vettore x in un vettore avente solo la prima componente diversa da 0.

$$Hx = -\sigma e_1 \quad \text{dove} \quad \sigma \in \mathbb{R} \text{ e } e_1 = (1, 0, 0, \ldots, 0)^T \in \mathbb{R}^{n \times 1}.$$

Per ricavare σ:

$$(Hx)^T(Hx) = (-\sigma e_1)^T(-\sigma e_1)$$
$$x^T \underbrace{H^T H}_{\text{I}} x = \sigma^2$$

Essendo H ortogonale:

$$x^T x = \sigma^2$$
$$\sigma^2 = ||x||_2^2$$

Ricaviamo addesso per quale w otteniamo la trasformazione $Hx = -\sigma e_1$:

$$Hx = (I - 2ww^T)x = x - 2ww^T x$$

definiamo $k = w^T x, k \in \mathbb{R}$ ottenendo:

$$x - 2ww^T x = x - 2wk$$

Abbiamo otttenuto $x - 2wk = -\sigma e_1$ in forma vettoriale:

$$\begin{pmatrix} x_1 - 2kw_1 \\ x_2 - 2kw_2 \\ \vdots \\ x_n - 2kw_n \end{pmatrix} = \begin{pmatrix} -\sigma \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

elevando entrambi i membri al quadrato:

$$\begin{pmatrix} 4k^2\omega_1^2 \\ 4k^2\omega_2^2 \\ \vdots \\ 4k^2\omega_n^2 \end{pmatrix} = \begin{pmatrix} \sigma^2 + x_1^2 + 2\sigma x_1 \\ x_2^2 \\ \vdots \\ x_n^2 \end{pmatrix}$$

e sommando opportunamente otteniamo:

$$4k^2 \underbrace{\left(\omega_1^2 + \omega_2^2 + \cdots + \omega_n^2\right)}_{\|\mathbf{w}\|_2^2=1} = \sigma^2 + 2\sigma x_1 + \underbrace{\left(x_1^2 + x_2^2 + \cdots + x_n^2\right)}_{\|\mathbf{x}\|_2^2=\sigma^2}$$

$$4k^2 = 2\sigma^2 + 2\sigma x_1 \tag{2}$$
$$2k^2 = \sigma(\sigma + x_1). \tag{3}$$

ponendo v=2kw ossia:

$$\mathbf{v} = \begin{pmatrix} 2kw_1 \\ 2kw_2 \\ \vdots \\ 2kw_n \end{pmatrix} = \begin{pmatrix} \sigma + x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \mathbf{x} + \sigma\mathbf{e}_1$$

ricaviamo:

$$H = I - 2\mathbf{w}\mathbf{w}^T = I - 2\frac{\mathbf{v}\mathbf{v}^T}{2k \cdot 2k} = I - \frac{\mathbf{v}\mathbf{v}^T}{2k^2} = I - \frac{\mathbf{v}\mathbf{v}^T}{\sigma(\sigma + x_1)}$$

Otteniamo quindi che per costruire una matrice elementare di Householder basta costruire un vettore $\mathbf{v}$ avente le stesse componenti del vettore $\mathbf{x}$ tranne la prima a cui si deve aggiungere $\sigma = \pm\|x\|_2$. Il segno di σ si sceglie concorde a quello di x_1 per limitare gli errori dovuti alla cancellazione numerica.

Banalmente non è necessario costruire la matrice H. Infatti generato $\mathbf{v} = (\sigma + x_1, x_2, \ldots, x_n)^T$ si ha

$$H\mathbf{x} = \mathbf{x} - \frac{\mathbf{v}\mathbf{v}^T\mathbf{x}}{\sigma(\sigma + x_1)}$$

a questo punto possiamo definire $\alpha = \mathbf{v}^T\mathbf{x} = \sum_{i=1}^{n} v_i x_i, \quad \beta = \frac{1}{\sigma(\sigma+x_1)}, \quad \tau = \alpha\beta$

$$Hx = \mathbf{x} - \tau\mathbf{v} = \begin{pmatrix} x_1 - \tau v_1 \\ x_2 - \tau v_2 \\ \vdots \\ x_n - \tau v_n \end{pmatrix}$$

Riusciamo in questo modo a ottenere Hx sottraendo dalla componente originale x_i il termine τv_i. Questa ottimizzazione ci permette di ottenere un costo computazionale di $O(n)$ inferiore al $O(n^2)$ del prodotto matrice vettore.

Uso delle matrici di Householder nella fattorizzazione QR

Idea di base: Utilizzare le matrici elementari di Householder per annullare gli elementi della matrice A che si trovano al di sotto della diagonale

Consideriamo dapprima la matrice H_1 che moltiplicata per A elimina le componenti del primo vettore colonna al di sotto del primo elemento, avremo $H_1\mathbf{a}_1 = -\sigma_1\mathbf{e}_1$ dove $\sigma_1 = \|\mathbf{a}_1\|_2$ e $\mathbf{v}_1 = \begin{pmatrix} \sigma_1 + a_{11} \\ a_{21} \\ \vdots \\ a_{n1} \end{pmatrix}$ ovviamente H_1 moltiplicherà tutte le colonne di A ottenendo:

$$H_1\mathbf{a}_i = \mathbf{a}_i - \tau^{(i)}\mathbf{v}_1, \qquad i = 2, 3, \ldots, n$$

$$\tau^{(i)} = \frac{\mathbf{v}_1^T\mathbf{a}_i}{\sigma_1(\sigma_1 + a_{11})}$$

$$H_1A = A^{(1)} = \begin{pmatrix} -\sigma_1 & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(1)} & \cdots & a_{nn}^{(1)} \end{pmatrix}$$

A questo punto consideriamo la matrice H_2 che moltiplicata per la matrice A_1 lascierà identiche prima riga e prima colonna modificherà il resto della matrice in particolare eliminando gli elementi sotto la diagonale della seconda colonna. Siano:

$$\mathbf{a}_2^{(1)} = \begin{pmatrix} a_{22}^{(1)} \\ a_{32}^{(1)} \\ \vdots \\ a_{n2}^{(1)} \end{pmatrix}, \quad \mathbf{a}_3^{(1)} = \begin{pmatrix} a_{23}^{(1)} \\ a_{33}^{(1)} \\ \vdots \\ a_{n3}^{(1)} \end{pmatrix}, \quad \ldots, \quad \mathbf{a}_n^{(1)} = \begin{pmatrix} a_{2n}^{(1)} \\ a_{3n}^{(1)} \\ \vdots \\ a_{nn}^{(1)} \end{pmatrix}$$

le colonne di $A^{(1)}$ a partire dalla seconda riga e tranne la prima colonna. Avremo $H_2\mathbf{a}_2^{(1)} = -\sigma_2\mathbf{e}_1$ dove $\sigma_2 = \pm\|\mathbf{a}_2^{(1)}\|_2$, mentre per le altre colonne si ha

$$\mathbf{v}_2 = \begin{pmatrix} \sigma_2 + a_{22}^{(1)} \\ a_{32}^{(1)} \\ \vdots \\ a_{n2}^{(1)} \end{pmatrix} \quad \Rightarrow \quad H_2\mathbf{a}_i^{(1)} = \mathbf{a}_i^{(1)} - \tau_i^{(1)}\mathbf{v}_2, \quad i = 3, \ldots, n$$

dove $\tau_i^{(1)} = \frac{\mathbf{v}_2^T\mathbf{a}_i^{(1)}}{\sigma_2(\sigma_2 + a_{22}^{(1)})}$ Al k-esimo passo avremo:

$$\mathbf{H}_k = \begin{pmatrix} \mathbf{I}_{k-1,k-1} & \mathbf{\Omega}_{k-1,n-k+1} \\ \mathbf{\Omega}_{n-k+1,k-1} & \overline{\mathbf{H}}_{n-k+1,n-k+1} \end{pmatrix}$$

dopo n-1 passi quindi per k=n-1 otteniamo:

$$\underbrace{H_{n-1}H_{n-2}\cdots H_3H_2H_1}_{=\mathbf{Q}^T}A = R$$

dove R è diagonale superiore, ricordando che Q è ortogonale arriviamo alla fattorizzazione A=QR.

Costo Computazionale

La fattorizzazione QR con le matrici di Householder è un metodo numericamente stabile, il costo computazionale dipende da due fattori, la costruzione delle matrici di Householder e l'applicazione di queste ultime. In particolare ogni trasformazione di Householder lavora su una sotto-matrice di A, con dimensioni che decrescono ad ogni iterazione mentre ad ogni trasformazione avremo un blocco di dimensioni (n-i+1)(n-i+1), in totale avremo:

$$\sum_{i=1}^{n} O((n-i+1)^2)$$

dove sostituendo con j=n-i+1 e applicando la formula della somma dei primi n quadrati $\sum_{i=0}^{n} i^2 = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ otteniamo:

$$O\left(\frac{2n^3}{3}\right)$$

3.2.4 Metodo QR

Il metodo QR ci permette di calcolare autovalori e autovettori di una matrice diagonalizzabile sfruttando la fattorizzazione QR. Definiamo un teorema che ci tornerà utile durante la spiegazione di questo metodo:

Teorema Due matrici simili A e $B = S^{-1}AS$ hanno gli stessi autovalori.

Dimostrazione:

$$\det(B-\lambda I) = \det(B-\lambda SS^{-1}) = \det(S^{-1}AS - \lambda SS^{-1}) =$$
$$\det[S^{-1}(A-\lambda I)S] = \det(S^{-1})\det(A-\lambda I)\det(S)$$

si verifica inoltre che se H_1 è simile a H_2 e H_2 è simile a H_3, allora H_1 è simile a H_3

Lemma Sia $A_0 = Q_0R_0$ ed $A_1 = R_0Q_0$. Le matrici A_0 e A_1 sono ortogonalmente simili e quindi hanno gli stessi autovalori, infatti otteniamo
$Q_0A_1Q_0^T = Q_0R_0Q_0Q_0^T = A_0$.

per chiarezza espositiva consideriamo la matrice $A_1 = A$

$$\begin{cases} A_1 = Q_1R_1 \\ A_2 = R_1Q_1 = Q_1^{-1}A_1Q_1 = Q_1^TA_1Q_1 \end{cases}$$

questo passaggio $A_2 = R_1Q_1 = Q_1^{-1}A_1Q_1 = Q_1^TA_1Q_1$ contiene la trasformazione $Q_1^TA_1Q_1$, essa rappresenta una trasformazione di similarità, di conseguenza le matrici A_1 e A_2 posseggono gli stessi autovalori, al passo successivo avremo:

$$\begin{cases} A_2 = Q_2R_2 \\ A_3 = R_2Q_2 = Q_2^{-1}A_2Q_2 = Q_2^TA_2Q_2 = Q_2^TQ_1^TA_1Q_1Q_2 = \\ \qquad (Q_1Q_2)^TA_1(Q_1Q_2) \end{cases}$$

da questa espressione $A_3 = R_2Q_2 = Q_2^{-1}A_2Q_2 = Q_2^TA_2Q_2 = Q_2^TQ_1^TA_1Q_1Q_2 = (Q_1Q_2)^TA_1(Q_1Q_2)$ possiamo vedere come A_3 può essere espressa in funzione delle matrici

Q_1,Q_2 e A_1.
Alla k-esima iterazione avremo quindi:

$$\begin{cases} A_k = Q_k R_k \\ A_{k+1} = R_k Q_k \end{cases}$$

dove A_{k+1} può essere anche espressa come:

$$A_{k+1} = (Q_1 Q_2 \dots Q_k)^T A_1 (Q_1 Q_2 \dots Q_k)$$

per il lemma visto all'inizio del paragrafo possiamo concludere che A_{k+1} è simile a A_k che è simile a A_{k-1},...,A_1, di conseguenza per il teorema definito precedentemente queste matrici posseggono gli stessi autovalori.

Convergenza del metodo QR

E' lecito chiedersi quando questo metodo converge:

Teorema Sia $A \in \mathbb{R}^{n \times n}$ una matrice con autovalori tutti distinti in modulo

$$|\lambda_1| > |\lambda_2| > \dots > |\lambda_n|$$

allora la successione $\{A_k = (a_{ij}^{(k)})\}_{k=1,2,\dots}$ converge ad una matrice triangolare superiore con $a_{ii}^{(k)} = \lambda_i$.
Nella pratica il processo si arresta quando

$$\max_{i>j} |a_{ij}^{(k)}| \leq \varepsilon, \quad \text{con } \varepsilon > 0 \text{ una tolleranza fissata.}$$

Lemma 1 Se $|\lambda_1| > |\lambda_2| > \dots > |\lambda_n|$ non è soddisfatta il metodo converge a una matrice quasi-triangolare, mentre se A è simmetrica allora la successione converge a una matrice diagonale.

Velocità di convergenza

Quando gli autovalori sono molto vicini, la convergenza è di solito lenta. Questo dipende del rapporto spettrale:

Definizione
Il rapporto spettrale è definito da:

$$\rho = \frac{|\lambda_{\text{max-1}}|}{|\lambda_{\text{max}}|}$$

$|\lambda_{\text{max-1}}|$ è il valore assoluto del secondo maggiore autovalore, mentre $|\lambda_{\text{max}}|$è il valore assoluto del maggiore autovalore.

Se ρ è piccolo, i valori al di fuori della diagonale decadono(vanno a 0) più velocemente rendendo la convergenza del metodo QR più rapida, mentre se $\rho = 1$ gli elementi fuori diagonale decadono più lentamente e la convergenza di QR richiede più tempo. Intuitivamente facendo anche una analogia coerente con il tema trattato: immaginiamo di voler separare due onde sonore, se queste ultime hanno frequenze molto diverse il sistema di separazione impiegherà poche iterazioni per dividerle, se invece le due onde hanno frequenze simili il sistema di separazione impiegherà molte iterate. In conclusione da questa analogia possiamo

sostenere che nel metodo QR gli autovalori vicini si comportano come le onde a frequenze simili richiedendo quindi più tempo per la convergenza del metodo.

Ottimizzare la velocità di convergenza

Una delle tecniche più utilizzate per aumentare la velocità di convergenza del metodo QR è chiamata "tecnica dello shift" , consiste nel:

$$\begin{cases} (A_k - t_k I) = Q_k R_k \\ A_{k+1} = R_k Q_k + t_k I \end{cases} \qquad \text{per } k = 1, 2, \dots$$

l'idea è quella di aggiungere uno scalare t_k sulla diagonale prima di fattorizzare. Si verifica facilmente che la successione generata con lo shift è simile alla matrice A:

$$\begin{aligned} A_{k+1} &= R_k Q_k + t_k I \\ &= Q_k^T Q_k \left(R_k Q_k + t_k I\right) \\ &= Q_k^T \left(Q_k R_k Q_k + t_k Q_k\right) \\ &= Q_k^T \left(Q_k R_k + t_k I\right) Q_k \\ &= Q_k^T A_k Q_k \\ &= Q_k^T \left(Q_{k-1}^T A_{k-1} Q_{k-1}\right) Q_k \\ &= \cdots \\ &= Q_k^T Q_{k-1}^T \cdots Q_1^T A_1 Q_1 Q_2 \cdots Q_k \\ &= \left(Q_1 \cdots Q_k\right)^T A_1 \left(Q_1 \cdots Q_k\right). \end{aligned}$$

generalmente si sceglie $t_k = a_{nn}^{(k)}$

Implementazione

Codice MATLAB:

```
function [T, Q, R] = qrgenerale(A, tol, kmax)
    [n, m] = size(A);


    if n ~= m
        error("La matrice deve essere quadrata");
    end


    T = A;


    test = norm(tril(A, -1), inf);


    n_iter = 0;


    while n_iter <= kmax && test > tol

        [Q, R] = qr(T);


        T = R * Q;


        n_iter = n_iter + 1;


        test = norm(tril(T, -1), inf);
    end


    if n_iter > kmax
        warning("Il metodo non converge nel massimo numero di
            iterazioni permesso\n");
    else

        fprintf("Il metodo converge in %d iterazioni\n", n_iter);
    end
end
```

Shift di Wilkinson

Una scelta efficace dello shift è scegliere come parametro l'autovalore della matrice

$$T[n-1:n, n-1:n] = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

più vicino a d.
Gli autovalori di $T[n-1:n, n-1:n]$ si trovano come radici del polinomio caratteristico:

$$\lambda_{1,2} = \frac{(a+d) \pm \sqrt{(a+d)^2 - 4(ad-bc)}}{2}$$

Tra i due autovalori si sceglie quello più vicino a d, ovvero quello che rende più piccola la differenza $|d - \lambda|$. Una volta trovato questo autovalore, lo shift si calcola come:

$$\mu = a + d \pm \Delta$$

Dove:

$$\Delta = \sqrt{(a+d)^2 - 4(ad-bc)}$$

Lo shift μ permette di modificare la matrice in modo che la trasformazione QR si concentri sugli elementi significativi, infatti, scegliendo un μ vicino a d, la differenza (d-μ) sarà molto piccola e il termine c diventa dominante nel sotto blocco, quindi le trasformazioni saranno mirate e il metodo QR convergerà più velocemente.

```
1
2  function mu = wilkinson_shift(submatrix)
3
4      if size(submatrix, 1) == 1
5          mu = submatrix;
6          return;
7      end
8
9      if size(submatrix, 1) == 2
10         a = submatrix(1,1);
11         b = submatrix(1,2);
12         c = submatrix(2,1);
13         d = submatrix(2,2);
14
15         % Calcolo del discriminante
16         disc = sqrt((a + d)^2 - 4 * (a*d - b*c));
17
18         % Selezione dello shift
19         if abs(a + d + disc) > abs(a + d - disc)
20             mu = a + d - disc;
21         else
22             mu = a + d + disc;
23         end
24     else
25         % Per matrici pi  grandi, usa l'ultimo elemento come
               shift
26         mu = submatrix(end, end);
27     end
28 end
```

Ottimizzazione metodo QR

Nel metodo QR a ogni passo si effettua una fattorizzazione QR, richiedendo quindi a ogni iterazione un costo $O(n^3)$. Una possibilità per ridurre il costo computazionale consiste nel trasformare la matrice A in forma di Hessenberg superiore, ovvero

$$t_{ij} = 0, \quad i > j + 1.$$

inoltre se A è simmetrica viene ridotta in forma tridiagonale

$$A = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ \vdots & \ddots & \ddots & & \\ & \ddots & \alpha_{n-1} & \beta_{n-1} \\ & & \beta_{n-1} & \alpha_n \end{pmatrix}$$

e calcolando gli autovalori otteniamo:

$$\lambda I - A = \begin{pmatrix} \lambda - \alpha_1 & -\beta_1 & & & \\ -\beta_1 & \lambda - \alpha_2 & -\beta_2 & & \\ & \ddots & \ddots & \ddots & \\ & & -\beta_{n-2} & \lambda - \alpha_{n-1} & -\beta_{n-1} \\ & & & -\beta_{n-1} & \lambda - \alpha_n \end{pmatrix}$$

si riconosce quella che è definita Successione di Sturm e possiamo quindi calcolare gli autovalori seguendo la successione:

$$\begin{cases} p_0(\lambda) = 1 \\ p_1(\lambda) = \lambda - \alpha_1 \\ p_i(\lambda) = (\lambda - \alpha_i)p_{i-1}(\lambda) - \beta_{i-1}^2 p_{i-2}(\lambda) \end{cases}$$

in questo caso, sfruttando la successione di Sturm e il teorema seguente:

Teorema Siano A una matrice tridiagonale reale e simmetrica con $\beta_i \neq 0$ per $i = 1, 2, ..., n-1$. Allora:

- Zeri reali e distinti: Gli zeri di $p_i(\lambda) = 0$ sono tutti reali e distinti.
- Alternanza degli zeri: Gli zeri di $p_i(\lambda) = 0$ si alternano con quelli di $p_{i-1}(\lambda) = 0$.

Inoltre, fissato $\gamma \in \mathbb{R}$ con $p_n(\gamma) \neq 0$, il numero di zeri di $p_n(\lambda) = 0$ che si trovano a destra di γ è uguale al numero di variazioni di segno nella successione:

$$1, p_1(\gamma), p_2(\gamma), ..., p_n(\gamma)$$

ignorando gli eventuali zeri.

conosciamo il numero di autovalori e inoltre utilizzando il teorema di Gershgorin è possibile restringere gli intervalli contenente gli autovalori, il procedimento è il seguente:

Si usa il teorema di Gerschgorin per localizzare gli zeri. Si individua un intervallo $[a_0, b_0]$ contenente tutti gli zeri di A e si sceglie $\gamma = \frac{a_0+b_0}{2}$.

Dal teorema si sa quanti sono gli zeri a destra e quanti sono gli zeri a sinistra di γ e si suddividono ulteriormente in due parti uguali gli intervalli $[a_0, \gamma]$ e $[\gamma, b_0]$, con

$$\gamma_1 = \frac{a_0 + \gamma}{2} \quad \text{e} \quad \gamma_2 = \frac{\gamma + b_0}{2}.$$

E così via, ripetendo questa tecnica di bisezione si riducono sempre di più le ampiezze degli intervalli, che tenderanno a zero e conterranno ognuno un autovalore.

Come avviene la trasformazione?

Consideriamo una matrice A simmetrica e applichiamo le matrici elementari di Householder, avremo quindi:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \quad \text{ed} \quad a_1 = \begin{pmatrix} a_{21} \\ \vdots \\ a_{n1} \end{pmatrix} \in \mathbb{R}^{n-1\times 1}$$

definiamo:

$$A = \begin{pmatrix} a_{11} & \mathbf{a}_1^T \\ \mathbf{a}_1 & \mathbf{A}_{11} \end{pmatrix}$$

all'iterazione k=1 avremo:

$$A^{(1)} \equiv A = \begin{pmatrix} a_{11} & \mathbf{a}_1^T \\ \mathbf{a}_1 & \mathbf{A}_{11} \end{pmatrix}$$
$$\bar{\mathbf{H}}_1\mathbf{a}_1 = \sigma_1\mathbf{e}_1 \in \mathbb{R}^{n-1\times 1}$$
$$A^{(2)} = \mathbf{H}_1 A^{(1)}\mathbf{H}_1$$

facendo i prodotti otteniamo:

$$A^{(2)} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & \bar{\mathbf{H}}_1 \end{pmatrix}}_{H_1} \underbrace{\begin{pmatrix} a_{11} & \mathbf{a}_1^T \\ \mathbf{a}_1 & \mathbf{A}_{11} \end{pmatrix}}_{A^{(1)}} \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & \bar{\mathbf{H}}_1 \end{pmatrix}}_{H_1}$$
$$= \begin{pmatrix} a_{11} & \mathbf{a}_1^T \\ \bar{\mathbf{H}}_1\mathbf{a}_1 & \bar{\mathbf{H}}_1\mathbf{A}_{11} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \bar{\mathbf{H}}_1 \end{pmatrix} = \begin{pmatrix} a_{11} & \mathbf{a}_1^T\bar{\mathbf{H}}_1 \\ \bar{\mathbf{H}}_1\mathbf{a}_1 & \bar{\mathbf{H}}_1\mathbf{A}_{11}\bar{\mathbf{H}}_1 \end{pmatrix}$$

A è simmetrica quindi: $\mathbf{a}_1^T\bar{\mathbf{H}}_1 = (\bar{\mathbf{H}}_1\mathbf{a}_1)^T = -\sigma_1\mathbf{e}_1^T \in \mathbb{R}^{1\times n-1}$ ottenendo:

$$A^{(2)} = \begin{pmatrix} a_{11} & (\bar{\mathbf{H}}_1\mathbf{a}_1)^T \\ \bar{\mathbf{H}}_1\mathbf{a}_1 & \bar{\mathbf{H}}_1\mathbf{A}_{11}\bar{\mathbf{H}}_1 \end{pmatrix}$$

iterando fino a n-1 otteniamo

$$A^{(n-1)} = \underbrace{H_{n-2}H_{n-3}\cdots H_1}_{H} A^{(1)} \underbrace{H_1H_2\cdots H_{n-2}}_{H^T}$$
$$= \begin{pmatrix} a_{11}^{(n-1)} & a_{21}^{(n-1)} & 0 & 0 & \cdots & 0 \\ a_{21}^{(n-1)} & a_{22}^{(n-1)} & a_{32}^{(n-1)} & 0 & \cdots & 0 \\ 0 & a_{32}^{(n-1)} & a_{33}^{(n-1)} & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & & 0 \\ 0 & \cdots & & & & \\ 0 & 0 & \cdots & \cdots & & a_{nn}^{(n-1)} \end{pmatrix}$$

Approfondimento matrici in forma di Hessenberg

Una matrice di Hessenberg è una matrice quadrata, che è quasi-triangolare. In particolare, una matrice di Hessenberg superiore ha valori pari a zero sotto la prima sotto-diagonale, e una matrice di Hessenberg inferiore li ha sopra la prima sovra-diagonale. La trasformazione in forma di Hessenberg di una matrice A conserva gli autovalori di quest'ultima.

$$A = PHP^T$$

Dove P è una matrice ortogonale. Banalmente la trasformazione di A in forma di Hessenberg, puo avvenire attraverso le trasformazioni di Householder, in particolare eliminano i termini al di sotto della prima sotto-diagonale.

3.2.5 Matrici di Givens

Le matrici elementari di Givens sono matrici di rotazione ortogonali e permettono di azzerare elementi di un vettore o di una matrice in modo selettivo.

Definizione: Fissati i e k e un angolo θ la matrice elementare di Givens è definita com:

$$G(i,k,\theta) = I_n - Y(i,k,\theta)$$

con

$$Y(i,k,\theta) \in \mathbb{R}^{n\times n}$$

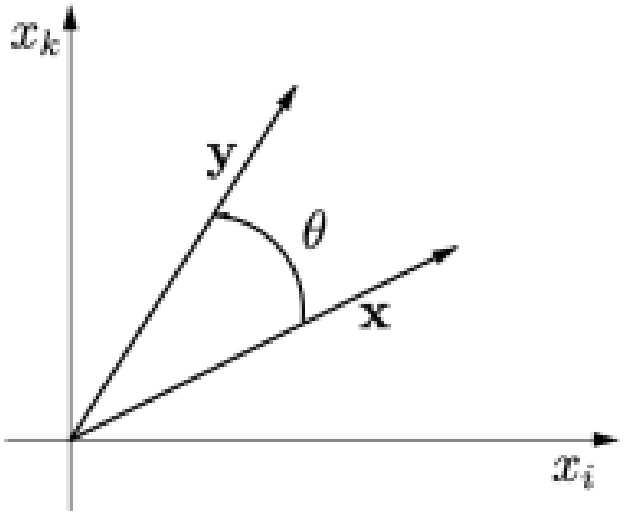
identicamente nulla, fatta eccezione per gli elementi:

$$y_{ii} = y_{kk} = 1 - \cos(\theta), \quad y_{ik} = -\sin(\theta) = -y_{ki}.$$

in forma esplicita:

$$G(i,k,\theta) = \begin{array}{c} \\ \\ i \\ \\ k \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccc} i & & & & k & \end{array} \\ \begin{pmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & \cos(\theta) & -\sin(\theta) & & \\ & & \sin(\theta) & \cos(\theta) & & \\ & & & & \ddots & \\ 0 & & & & & 1 \end{pmatrix} \end{array}$$

Il prodotto $y = G_{ij}x$ dove x è un generico vettore, corrisponde a ruotare in senso antiorario x di un angolo θ nel piano (x_i, x_j).



e risulta:

$$y_k = \begin{cases} x_k & k \neq i, j \\ \cos(\theta)x_i - \sin(\theta)x_j & k = i \\ \sin(\theta)x_i + \cos(\theta)x_j & k = j \end{cases}$$

Una matrice di Givens G_{ij} con $i = 1, \ldots, n-1$ e $j = i+1, \ldots, n$ applicata a una matrice A annulla l'elemento di posto (i, j) dove:

$$\cos\theta = \frac{a_{ii}}{\sqrt{a_{ii}^2 + a_{ij}^2}}$$
$$\sin\theta = -\text{sign}(a_{ij})\frac{a_{ij}}{\sqrt{a_{ii}^2 + a_{ij}^2}}$$

L'utilizzo delle matrici di Givens ci permette di porre a zero elementi in modo più selettivo. Nel caso delle matrici in forma di Hessenberg ci basta annullare la diagonale al di sotto della diagonale principale per effettuare la fattorizzazione QR. Riduciamo quindi la matrice iniziale A in forma di Hessenberg e poi effettuiamo la fattorizzazione QR utilizzando le matrici di Givens.

Riepilogando la matrice G_{ij} annulla l'elemento (i, j) di una matrice A, quindi ci basta moltiplicare $n - 1$ matrici di Givens che corrispondono al numero di elementi della sottodiagonale.

Implementazione

Codice Matlab:

```
function [Q, R] = qrgivensnostra(A)
    [m, n] = size(A);
    R = A;
    Q = eye(m);

    for j = 1 : n
        % Se la sotto-diagonale esiste, cioe' se j+1 <= m
        if j+1 <= m
            % L'unico elemento che potrebbe essere non nullo e' R(
                j+1, j)
            a = R(j,   j);
            b = R(j+1, j);

            [c, s] = calcolo_parametri_givens_progetto(a, b);
            G = [c -s; s c];

            % Aggiorna le 2 righe di R
            R([j, j+1], j:end) = G * R([j, j+1], j:end);

            % Aggiorna Q
            Q(:, [j, j+1]) = Q(:, [j, j+1]) * G';
        end
    end
end


function [c, s] = calcolo_parametri_givens_progetto(a, b)
    if b == 0
        c = 1;
        s = 0;
    else
        if abs(b) > abs(a)
            tau = -a / b;
            s = 1 / sqrt(1 + tau^2);
            c = s * tau;
        else
            tau = -b / a;
            c = 1 / sqrt(1 + tau^2);
            s = c * tau;
        end
    end
end
```

3.2.6 Metodo QR utilizzando Hessenberg, Householder e Givens

Per applicare il metodo QR una delle tecniche più utilizzata è quella di ridurre inizialmente la matrice A in forma di Hessenberg, attraverso le matrici di Householder e a ogni passo A_k fattorizzare utilizzando le matrici di Givens, che agiscono molto bene sulla struttura sparsa della matrice di Hessemberg. Con questo metodo è possibile ovviamente ricavare gli autovalori che si trovano sulla diagonale della matrice finale ma anche gli autovettori, in particolare definiamo una matrice V che contiene l'accumulo delle trasformazioni ortogonali Q_k nel dettaglio:

$$V = Q_1 Q_2 \cdots Q_k$$

Ricordando adesso la trasformazione in forma di Hessenberg:

$$A = PHP^T$$

Quindi abbiamo la matrice P che tiene traccia delle rotazioni ortogonali applicate durante l'algoritmo di Hessenberg e QR. Ogni colonna di P è una rotazione ortogonale che ha trasformato la matrice A in una forma di Hessenberg e la matrice V che accumula gli autovettori relativi alla matrice di Hessenberg H durante il processo QR. In altre parole, le colonne di V sono gli autovettori rispetto alla matrice H, ma non ancora rispetto alla matrice A originale.

Quindi, moltiplicando P (che contiene le trasformazioni ortogonali) per V (che contiene gli autovettori di H), otteniamo gli autovettori di A.

$$\text{autovettori} = P \cdot V;$$

Analizziamo il costo computazionale:

- Riduzione di A in forma di Hessenberg: il costo per questa operazione è pari a $O(n^3)$, ma viene eseguita solo una volta all'inizio.
- Applicazione fattorizzazione QR per ogni iterazione: in particolare in ogni iterazione avviene una fattorizzazione QR utilizzando il metodo di Givens, il costo computazionale dell'algoritmo di Givens è pari a $O(n^3)$ in questo caso però applichiamo Givens a matrici di Hessenberg, il costo si riduce quindi a $O(n^2)$ perché il numero degli elementi da azzerare è n

implementazione:

```
function [eigenvalues, eigenvectors] = qr_hessenberg_shift(A)

    max_iterations = 10000;
    tol = 1e-5;

    % Riduzione a forma di Hessenberg
    [P, H] = hess(A);

    % Inizializzazione
    n = size(H, 1);
    V = eye(n);  % Matrice per accumulare gli autovettori

    % Ciclo principale QR con shift
    for iter = 1:max_iterations
        % Trovare il primo sottodiagonale significativo
        m = n;
        while m > 1
            if abs(H(m, m-1)) < tol * (abs(H(m-1, m-1)) + abs(H(m,
                m)))
                m = m - 1;
            else
                break;
            end
        end
        % Shift di Wilkinson
        mu = wilkinson_shift(H(max(1,m-1):m, max(1,m-1):m));
        % Decomposizione QR con shift
        [Q, R] = qrgivensnostra(H - mu * eye(n));
        % Aggiornamento matrice di Hessenberg e autovettori
        H_new = R * Q + mu * eye(n);
        V = V * Q;
        % Verificare la convergenza confrontando la nuova e
            vecchia matrice
        if norm(H - H_new, 'inf') < tol
            break;
        end
        H = H_new;
    end

    % Estrazione degli autovalori
    eigenvalues = diag(H);
    % Ricostruzione degli autovettori
    eigenvectors = P * V;
    % Normalizzazione degli autovettori
    for i = 1:n
        eigenvectors(:,i) = eigenvectors(:,i) / norm(eigenvectors
            (:,i));
    end
end
```

3.3 Implementazione SVD

```
function [U, S, V] = svd_qr_hessenberg_shift(A)
    AtA = A' * A;

    [eigvals, eigvecs] = qr_hessenberg_shift(AtA);

    singular_values = sqrt(abs(eigvals));

    [sorted_singular_values, idx] = sort(singular_values, 'descend');

    V = eigvecs(:, idx);

    S = diag(sorted_singular_values);

    U = A * V / S;

    % Normalizzo U
    for i = 1:size(U, 2)
        U(:, i) = U(:, i) / norm(U(:, i));
    end
end
```

3.4 Trasformata discreta di Fourier

La trasformata di Fourier è uno strumento molto potente che ci consente di analizzare un segnale nel dominio delle frequenze. Per quel che riguarda il nostro progetto, useremo la trasformata per effettuare l'analisi spettrale dei risultati ottenuti mettendoli a confronto con quelli del segnale originario.

3.4.1 Trasformata di Fourier tramite prodotto matrice vettore

Nel continuo definiamo la trasformata di Fourier di una funzione $f(x)$ come:

$$\hat{f}(p) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi px}\, dx$$

E la rispettiva antitrasformata che ci consente di tornare nel dominio di partenza:

$$f(x) = \int_{-\infty}^{\infty} \hat{f}(p) e^{i2\pi px}\, dp$$

Dati n punti equidistanti e i rispettivi valori assunti dalla funzione in questi punti:

$$\begin{cases} f(x_k) \\ x_k = k\Delta x \end{cases} \quad k = 0, ..., n-1$$

La trasformata discreta di Fourier ci restituisce n valori nel dominio delle frequenze:

$$\begin{cases} \hat{f}(p_j) \\ p_j = \frac{j}{n\Delta x} \end{cases} \quad j = 0, ..., n-1$$

Nel continuo, nel calcolo della trasformata, l'integrale diventa una sommatoria e la formula si trasforma come segue:

$$\hat{f}(p_j) = \sum_{k=0}^{n-1} f(x_k) e^{-i2\pi p_j x_k} \Delta x = \Delta x \sum_{k=0}^{n-1} f(x_k) e^{-i2\pi \frac{j}{n\Delta x} k\Delta x} = \Delta x \sum_{k=0}^{n-1} f(x_k) e^{-i\frac{2\pi}{n} jk}$$

Facendo uso delle radici $n-esime$ dell'unità:

$$\omega_n = \cos\frac{2\pi}{n} + i \sin\frac{2\pi}{n} = e^{i\frac{2\pi}{n}}$$

$$\hat{f}(p_j) = \Delta x \sum_{k=0}^{n-1} f(x_k) e^{-i\frac{2\pi}{n} jk} = \Delta x \sum_{k=0}^{n-1} f(x_k) w_n^{-jk}$$

Al variare di k e j:

$$\Delta x \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega_n^{-1} & \cdots & \omega_n^{-(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-j} & \cdots & \omega_n^{-(n-1)j} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \cdots & \omega_n^{-(n-1)^2} \end{bmatrix} \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{n-1}) \end{bmatrix} = \begin{bmatrix} \hat{f}(p_0) \\ \hat{f}(p_1) \\ \vdots \\ \hat{f}(p_{n-1}) \end{bmatrix}$$

Ovvero applicando la matrice delle radici n-esime dell'unità al vettore delle valutazioni di f otteniamo il vettore delle valutazioni delle trasformate. Il costo di questo metodo è $O(n^2)$, ovvero il costo del prodotto matrice-vettore.

Possiamo fare considerazioni analoghe sull'antitrasformata:

$$f(x_k) = \sum_{j=0}^{n-1} \hat{f}(p_j) e^{i\frac{2\pi}{n}jk} \Delta p = \Delta p \sum_{j=0}^{n-1} \hat{f}(p_j) \omega_n^{jk}$$

Al variare di k e j:

$$\Delta p \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega_n^1 & \cdots & \omega_n^{(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^j & \cdots & \omega_n^{(n-1)j} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} \begin{bmatrix} \hat{f}(p_0) \\ \hat{f}(p_1) \\ \vdots \\ \hat{f}(p_{n-1}) \end{bmatrix} = \begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{n-1}) \end{bmatrix}$$

Anche per l'antitrasformata il costo computazionale risulta essere $O(n^2)$

Implementazione

Trasformata tramite prodotto matrice-vettore:

```
function F = DFT_via_matrix(f)


    % Numero di campioni
    n = length(f);

    % Creazione della matrice DFT
    omega_n = exp(-2i * pi / n); % Radice n-esima dell'unita'
    k = (0:n-1)'; % Indici delle righe
    j = 0:n-1;    % Indici delle colonne
    W = omega_n .^ (k * j); % Matrice DFT, k*j e' una matrice
        [0,...,0; 0, 1, 2,...,n-1;...]

    % Prodotto matrice-vettore
    F = W * f(:);
end
```

Implementazione

Antitrasformata tramite prodotto matrice-vettore:

```
function f = IDFT_via_matrix(F)

    % Numero di campioni
    n = length(F);

    % Creazione della matrice IDFT
    omega_n = exp(2i * pi / n); % Radice n-esima dell'unita
    k = (0:n-1)'; % Indici delle righe
    j = 0:n-1;    % Indici delle colonne
    W_inv = omega_n .^ (k * j); % Matrice IDFT

    % Prodotto matrice-vettore con normalizzazione
    f = (1/n) * (W_inv * F(:));
end
```

3.4.2 Trasformata veloce di Fourier

La trasformata veloce di Fourier ci consente di ottenere la trasformata con un costo minore pari a $O(nlog(n))$.
Posto:

$$f_k := f(x_k) \quad k = 0, ..., n-1$$

Distinguiamo l'insieme delle valutazioni con indice pari e dispari, ognuno dei quali conterrà $M = \frac{n}{2}$ elementi:

$$\begin{cases} f_0, f_2, f_4, ..., f_{n-2} \\ f_1, f_3, f_5, ..., f_{n-1} \end{cases}$$

Ora consideriamo la trasformata di Fourier dei dati con indice pari e di quelli con indice dispari:

$$\hat{f}_j^{(1)} = \sum_{k=0}^{M-1} f_{2k}\omega_M^{-jk} = f_0 + f_2\omega_M^{-j} + ... + f_{n-2}\omega_M^{-(M-1)j}$$

$$\hat{f}_j^{(2)} = \sum_{k=0}^{M-1} f_{2k+1}\omega_M^{-jk} = f_1 + f_3\omega_M^{-j} + ... + f_{n-1}\omega_M^{-(M-1)j}$$

Ma:

$$\omega_M = e^{i\frac{2\pi}{M}} = e^{i\frac{2\pi}{\frac{n}{2}}} = \omega_n^2$$

E dunque le due trasformate si modificano come segue:

$$\hat{f}_j^{(1)} = \sum_{k=0}^{M-1} f_{2k}\omega_n^{-2jk} = f_0 + f_2\omega_n^{-2j} + ... + f_{n-2}\omega_n^{-2(M-1)j}$$

$$\hat{f}_j^{(2)} = \sum_{k=0}^{M-1} f_{2k+1}\omega_n^{-2jk} = f_1 + f_3\omega_n^{-2j} + ... + f_{n-1}\omega_n^{-2(M-1)j}$$

Per definizione la trasformata di Fourier è:

$$\hat{f}_j = \sum_{k=0}^{n-1} f_k \omega_n^{-jk} = f_0 + f_1\omega_n^{-j} + f_2\omega_n^{-2j} + f_3\omega_n^{-3j} + \dots$$

Da cui si nota come alla $\hat{f}_j^{(2)}$ manchi un fattore moltiplicativo ω_n^{-j}, e quindi:

$$\hat{f}_j = \hat{f}_j^{(1)} + \omega_n^{-j}\hat{f}_j^{(2)} \quad j = 0, ..., M-1$$

Si osserva però che in questo modo vengono prodotti solo $\frac{n}{2}$ elementi della trasformata, l'altra metà la posso ottenere come:

$$\hat{f}_{j+M} = \hat{f}_j^{(1)} - \omega_n^{-j}\hat{f}_j^{(2)} \quad j = 0, ..., M-1$$

Infatti:

$$\hat{f}_{j+M} = \sum_{k=0}^{n-1} f_k \omega_n^{-(j+M)k}$$

Sviluppando il termine $\omega_n^{-(j+M)k}$:

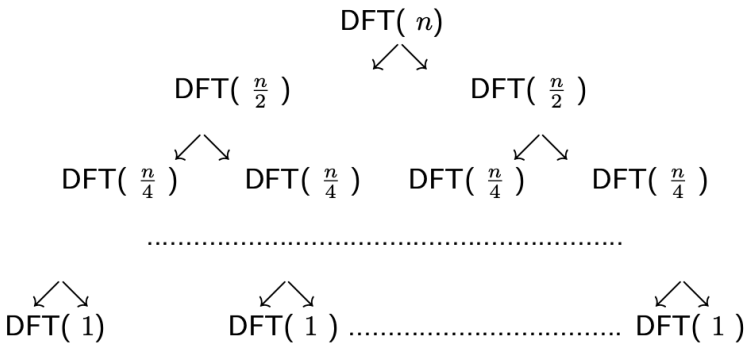
$$\omega_n^{-(j+M)k} = \omega_n^{-jk}\omega_n^{-Mk} = \omega_n^{-jk}\omega_n^{-\frac{n}{2}k} = \omega_n^{-jk}e^{i(\frac{2\pi}{n})\frac{-n}{2}k} = \omega_n^{-jk}e^{i\pi k} = (-1)^k w_n^{-jk}$$

$$\hat{f}_{j+M} = \sum_{k=0}^{n-1}(-1)^k f_k\omega_n^{-jk} = f_0 - f_1\omega_n^{-j} + f_2\omega_n^{-2j} - f_3\omega_n^{-3j} + \dots = \hat{f}_j^{(1)} - \omega_n^{-j}\hat{f}_j^{(2)}$$

Dunque si possono sfruttare due sottotrasformate $\hat{f}_j^{(1)}$ e $\hat{f}_j^{(2)}$ di ordine $\frac{n}{2}$ per ottenere l'intera trasformata di ordine n.

$$\begin{cases} \hat{f}_j = \hat{f}_j^{(1)} + \omega_n^{-j}\hat{f}_j^{(2)} \\ \hat{f}_{j+M} = \hat{f}_j^{(1)} - \omega_n^{-j}\hat{f}_j^{(2)} \end{cases} \quad j = 0, ..., M-1$$

Generalizzando possiamo procedere ricorsivamente andando sempre a dimezzare il numero di elementi su cui lavorare, generando cosi un albero binario i cui nodi foglia sono le trasformate di un singolo valore:



Lavorando in questo modo otteniamo un costo computazionale di $O(nlog(n))$ poichè $log(n)$ rappresenta l'altezza dell'albero binario mentre n il numero di operazioni che svolgiamo per ogni livello.

Implementazione

Trasformata veloce di Fourier:

```
function F = trasformata_veloce(f)


    n = length(f);

    % Padding del vettore f con zeri fino a farne una lunghezza
        che sia una potenza di 2
    next_pow2 = 2^nextpow2(n); % Trova la prossima potenza di 2
    f = [f, zeros(1, next_pow2 - n)]; % Aggiungi gli zeri

    n = length(f); % Ora n e' una potenza di 2

    % Caso base: se n e' 1, la FFT e' il vettore stesso
    if n == 1
        F = f;
        return;
    end

    % Radice n-esima dell'unita'
    omega_n = exp(-2i * pi / n);

    % Dividi il vettore in due parti: indici pari e dispari
    f_1 = f(1:2:end); % Componenti con indici pari
    f_2 = f(2:2:end);  % Componenti con indici dispari

    % Calcola ricorsivamente la FFT sui sottoinsiemi
    F_1 = trasformata_veloce(f_1);
    F_2 = trasformata_veloce(f_2);

    % Utilizzo le formule
    F = zeros(1, n);
    for j = 0:(n/2-1)
        F(j+1) = F_1(j+1) + omega_n^j * F_2(j+1); % Parte
            inferiore
        F(j+1+n/2) = F_1(j+1) - omega_n^j * F_2(j+1); % Parte
            superiore
    end
end
```

3.5 Potenza di un segnale digitale

È necessario introdurre il concetto di potenza di un segnale digitale, che ci sarà utile nell'algoritmo nella selezione automatica del parametro k che vedremo in seguito. Concettualmente, la potenza di un segnale è un indice che a differenza dell'energia è indipendente dalla finestra di osservazione del segnale. In particolare, la potenza di un segnale digitale è definita come:

$$P = \frac{1}{N} \sum_{n=1}^{N} |x(n)|^2$$

dove $x(n)$ è il valore del segnale nel campione n, N è il numero totale dei campioni, mentre $|x(n)|^2$ è la potenza istantanea di un segnale in un punto specifico nel tempo.

3.5.1 Potenza e Energia del Segnale Digitale

La potenza di un segnale digitale è una misura della sua intensità media nel tempo ed è fondamentale per comprendere il comportamento del segnale durante i processi di elaborazione. Mentre l'energia di un segnale si riferisce alla quantità totale di "intensità" contenuta in tutto il segnale, la potenza è una quantità che dipende dal tempo di osservazione ed è particolarmente utile quando si analizzano segnali che si evolvono nel tempo, come nel caso del trattamento audio.

La potenza, come definita dalla formula:

$$P = \frac{1}{N} \sum_{n=1}^{N} |x(n)|^2$$

è una media dei quadrati dei valori del segnale. Questa formula assume che il segnale sia discreto, cioè campionato a intervalli regolari di tempo.

L'energia totale di un segnale $x(n)$ è calcolata come la somma dei quadrati di tutti i suoi campioni:

$$E = \sum_{n=1}^{N} |x(n)|^2$$

L'energia del segnale descrive l'intensità totale, mentre la potenza è legata alla durata del segnale ed è quindi utile quando si lavora con segnali che sono osservati per un intervallo di tempo.

3.5.2 Significato della Potenza nei Processi di Elaborazione

Nel contesto dell'elaborazione del segnale, la potenza è utile nella riduzione del rumore, l'obiettivo è quello di mantenere la potenza del segnale utile (il segnale originale) mentre si rimuove il rumore indesiderato. La potenza del segnale denoised (segnale ridotto) è quindi un indicatore chiave per determinare l'efficacia del processo di riduzione del rumore.

3.5.3 Potenza Media e Potenza Istantanea

La potenza istantanea di un segnale in un campione n è definita come il quadrato del valore assoluto del campione stesso:

$$P_{\text{ist}}(n) = |x(n)|^2$$

Questa misura da un'idea di quanto sia intensa la variazione del segnale in un determinato momento.

Inoltre la potenza media è una misura più stabile nel tempo e si ottiene dalla media della potenza istantanea su un intervallo di tempo, mentre la potenza istantanea fornisce informazioni su come l'intensità cambia nel tempo.

3.5.4 Potenza e Qualità del Segnale

In generale, la potenza del segnale denoised deve essere simile o inferiore alla potenza del segnale originale per garantire una buona qualità del segnale. Se la potenza denoised è troppo alta, potrebbe esserci una distorsione introdotta nel segnale, mentre se è troppo bassa il segnale potrebbe risultare troppo smorzato, un analisi approfondita verrà eseguita nell'implementazione.

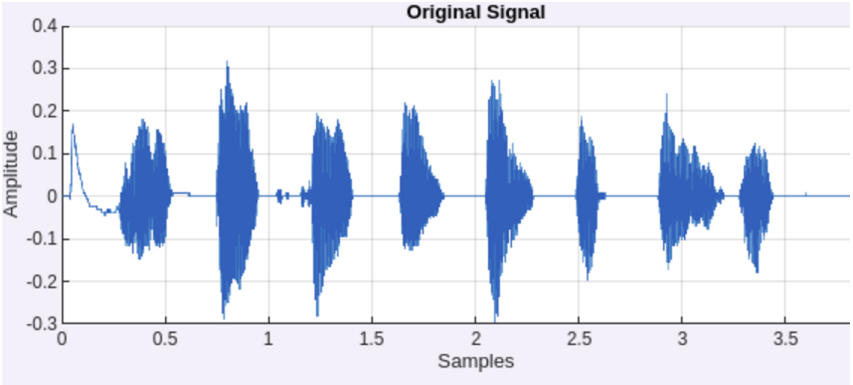
4 Implementazione dell'algoritmo

4.1 Creazione della matrice audio

4.1.1 Caricamento del file audio e inserimento del rumore

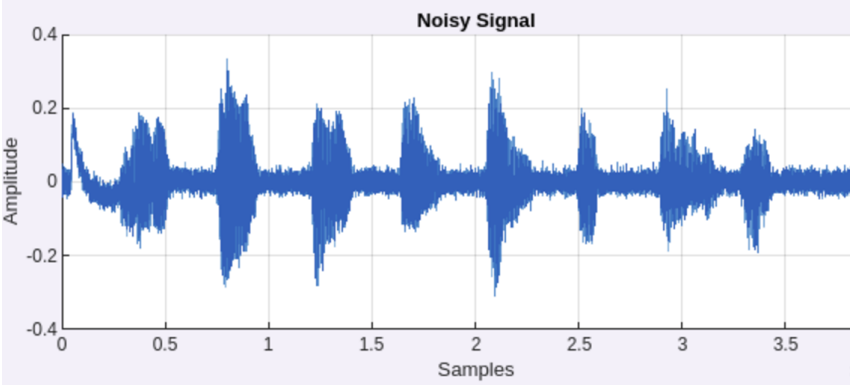
```
1
2       [x, fs] = audioread('voice.wav');
3       if size(audio, 2) == 2
4           audio = mean(audio, 2);
5       end
6       sound(x, fs);
7       plot(app.OriginalSignalAxes, x);
8       audio = awgn(x,10,'measured');
9       filename = 'noisyvoice.wav';
10      audiowrite(filename, audio, fs)
11      pause(5)
12      sound(audio, fs);
13      plot(app.NoisySignalAxes, audio);
```

Il file audio viene caricato tramite la funzione **audioread**, che crea un vettore x contenente il segnale audio campionato, in caso di audio stereo questo viene convertito in mono facendo la media dei due canali; l'audio mono utilizza un solo canale per registrare e riprodurre il suono e dunque ogni altoparlante (destro o sinistro) riproduce lo stesso segnale, mentre l'audio stereo utilizza due canali diversi, uno per il lato sinistro e uno per il lato destro, che possono contenere informazioni diverse per dare un effetto spaziale al suono. Nel nostro caso poiché cerchiamo di pulire un segnale, la maggior parte delle informazioni si troveranno già in un unico canale, mentre il secondo serve per aggiungere effetti stereo che non sono importanti per il fine del nostro progetto e possiamo ignorarli; inoltre lavorare con un segnale mono riduce la complessità computazionale poiché ci consente di lavorare su una sola matrice audio.
Tramite la funzione **sound** possiamo ascoltare il segnale, mentre con **plot(app.OriginalSignalAxes,x)** otteniamo la seguente rappresentazione temporale del segnale:



Successivamente tramite la funzione **awgn(x,10,'measured')** applichiamo al segnale audio x un **rumore bianco gaussiano**, simulando un audio degradato; il secondo parametro della funzione indica il rapporto segnale-rumore espresso in decibel, un valore più alto indica più rumore, uno più basso meno rumore. Il terzo parametro **'measured'** indica che l'energia del segnale originale viene calcolata automaticamente per stimare quanto rumore aggiungere.

In particolar modo abbiamo scelto di applicare un rumore gaussiano poiché la SVD funziona molto bene con questa tipologia di rumore dato che tende a distribuire la sua energia in tutte le componenti della matrice, ma è particolarmente influente sulle componenti a basso valore singolare che andiamo a tagliare con l'SVD, inoltre è uno dei rumori più comune.
Infine salviamo il segnale audio col rumore in un nuovo file e lo plottiamo ottenendo il seguente grafico temporale:



4.1.2 Suddivisione in finestre e creazione della matrice

```
frameLength = 1024;
overlap = 512;
hopSize = frameLength - overlap;


numFrames = floor((length(audio) - overlap) / hopSize)
audioMatrix = zeros(frameLength, numFrames);
for i = 1:numFrames
    startIdx = (i - 1) * hopSize + 1;
    endIdx = startIdx + frameLength - 1;
    audioMatrix(:, i) = audio(startIdx:endIdx);
end
```

Per lavorare con l'audio in modo efficiente lo dividiamo in **finestre**, ogni finestra è un frammento di audio su un piccolo intervallo di tempo. In particolare si usano finestre sovrapposte, ovvero una finestra non inizia precisamente da dove finisce la precedente ma prima che la precedente finisca; la sovrapposizione tra finestre permette di catturare meglio le transizione nell'audio, come un cambio di tono o di timbro, rispetto a finestre contigue, in questo modo la transizione da un frammento all'altro è più fluida.
Nel codice abbiamo quindi stabilito i parametri della finestra:

- **frameLength = 1024**, ovvero ogni finestra è composta da 1024 campioni prelevati dal vettore del segnale.
- **overlap = 512**, ogni finestra inizia 512 campioni prima che finisca la precedente
- **hopSize = frameLength - overlap**, è un parametro che rappresenta la distanza tra l'inizio di una finestra e l'inizio della finestra successiva.

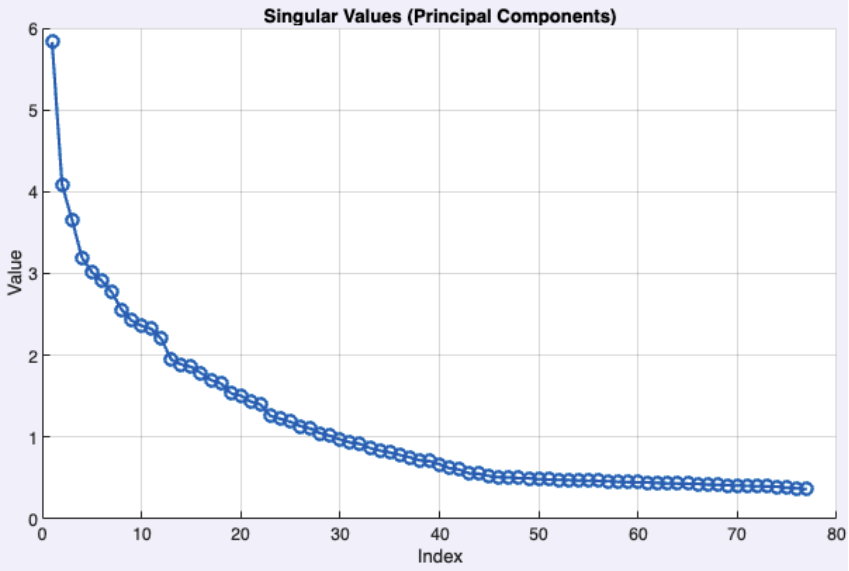
Procediamo al calcolo del numero di finestre con **numFrames = floor((length(audio) - overlap) / hopSize)**, in particolare **length(audio) - overlap** rappresenta il numero di

campioni effettivamente utilizzabili dopo aver tenuto conto dell'overlap, dividendo per **hopSize** determiniamo quante finestre possiamo calcolare in totale dato il passo tra le finestre; infine la funzione **floor** arrotonda il numero di finestre al numero intero più vicino verso il basso, dato che non possiamo avere un numero di finestre non intero.
Nella matrice ogni colonna rappresenta una finestra e ogni riga un campione.
Nella costruzione della matrice determiniamo l'indice di partenza di una finestra come **startIdx = (i - 1) * hopSize + 1** e l'indice finale come **endIdx = startIdx + frameLength - 1**, infine **audioMatrix(:, i) = audio(startIdx:endIdx)** inserisce la finestra estratta come colonna i-esima della matrice.

4.2 Applicazione della SVD sulla matrice audio

```
1
2       [U, S, V] = svd_qr_hessenberg_shift(audioMatrix);
3       % Estrazione dei valori singoli
4       singularValues = diag(S);
5       plot(app.Valorisingolari,singularValues, 'o-', 'LineWidth',
            1.5, 'MarkerSize', 6);
```

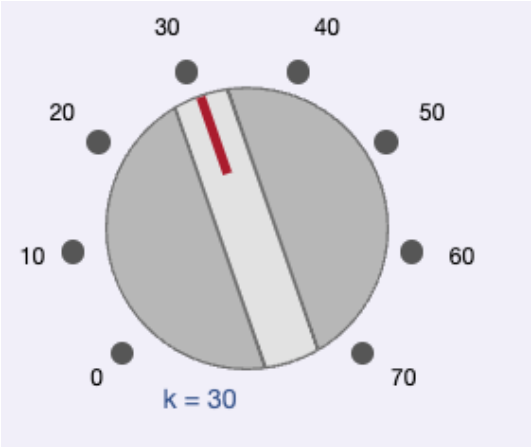
Tramite la funzione **SVD** che abbiamo costruito nella sezione "Fondamenti Teorici: Implementazione SVD" siamo in grado di ricavare le matrici U, Σ e V della decomposizione ai valori singolari. Successivamente estraiamo tutti i valori singolari dalla matrice Σ e li visualizziamo tramite il plot, ottenendo il seguente grafico:



La selezione del parametro K, ovvero dei valori singolari che si vogliono preservare, è molto sensibile poiché un K troppo piccolo rimuoverebbe una quantità eccessiva di informazioni, portando a una qualità audio molto bassa, pur eliminando gran parte del rumore, mentre un K troppo elevato preserva troppe informazioni del segnale originale, riducendo l'efficacia nella rimozione del rumore.
Dunque abbiamo implementato la possibilità di effettuare una scelta del parametro K manuale o automatica.

4.2.1 Selezione del parametro k manualmente

Abbiamo implementato la possibilità di effettuare una selezione del parametro K manuale, tramite interfaccia grafica, così da poter verificare empiricamente come differisce l'audio finale in base al parametro K scelto.



4.2.2 Selezione del parametro k automatica

Abbiamo sviluppato un algoritmo che permette di scegliere il k in base alle informazioni contenute nei valori singolari e alla potenza del segnale originale e del segnale audio.

```
originalPower = sum(audio.^2) / length(audio);
```

inizialmente calcoliamo la potenza del segnale originale utilizzando la formula definita nella sezione "Fondamenti teorici: Potenza di un segnale digitale", a questo punto fissiamo un k che permette di mantenere il 50% delle informazioni:

```

    cumulativeInformation = 0;
    k_initial = 1;

    while cumulativeInformation < 50
        singularValue = S(k_initial, k_initial);
        cumulativeInformation = cumulativeInformation + (
            singularValue / totalSingularValueSum) * 100;
        k_initial = k_initial + 1;
        if k_initial > k_max
            break;
        end
    end
```

Questo ci permette di garantire che il segnale ricostruito con i primi k valori singolari contenga almeno metà dell'informazione originale. Ora calcoliamo la potenza del segnale denoised basato sul k precedentemente scelto(quello che mantiene il 50% delle informazioni)

```
    % Calcola la potenza del segnale denoised per k iniziale
    U_com = U(:, 1:k);
    S_com = S(1:k, 1:k);
    V_com = V(:, 1:k);

    audioMatrix_denoised = U_com * S_com * V_com';
    audio_denoised = zeros(length(audio), 1);

    for i = 1:numFrames
        startIdx = (i - 1) * hopSize + 1;
        endIdx = min(startIdx + frameLength - 1, length(audio));
        audio_denoised(startIdx:endIdx) = audio_denoised(startIdx:
            endIdx) + audioMatrix_denoised(:, i);
    end

    denoisedPower = sum(audio_denoised.^2) / length(audio_denoised
        );
```

Una volta ricavata la potenza del segnale denoised useremo quest'ultima per scegliere il valore di k finale, in particolare:

```

    if denoisedPower < originalPower
        while denoisedPower < originalPower && k < k_max
            k = k + 1;
            U_com = U(:, 1:k);
            S_com = S(1:k, 1:k);
            V_com = V(:, 1:k);

            audioMatrix_denoised = U_com * S_com * V_com';
            audio_denoised = zeros(length(audio), 1);
            for i = 1:numFrames
                startIdx = (i - 1) * hopSize + 1;
                endIdx = min(startIdx + frameLength - 1, length(
                    audio));
                audio_denoised(startIdx:endIdx) = audio_denoised(
                    startIdx:endIdx) + audioMatrix_denoised(:, i);
            end

            denoisedPower = sum(audio_denoised.^2) / length(
                audio_denoised);
            if k > floor(0.65 * k_max)
                break;
            end
        end
    elseif denoisedPower > 1.1 * originalPower
        while denoisedPower > 1.1 * originalPower && k > ceil(0.3
            * k_max)
            k = k - 1;
            U_com = U(:, 1:k);
            S_com = S(1:k, 1:k);
            V_com = V(:, 1:k);

            audioMatrix_denoised = U_com * S_com * V_com';
```

```
30              audio_denoised = zeros(length(audio), 1);
31              for i = 1:numFrames
32                  startIdx = (i - 1) * hopSize + 1;
33                  endIdx = min(startIdx + frameLength - 1, length(
                        audio));
34                  audio_denoised(startIdx:endIdx) = audio_denoised(
                        startIdx:endIdx) + audioMatrix_denoised(:, i);
35              end
36
37              denoisedPower = sum(audio_denoised.^2) / length(
                    audio_denoised);
38              if k < ceil(0.3 * k_max)
39                  break;
40              end
41          end
42      end
```

In questo frammento di codice troviamo il valore ottimale di k che bilancia la quantità di informazioni preservate nel segnale e la potenza del segnale denoised rispetto a quella del segnale originale.

Quando la potenza del segnale denoised risulta inferiore a quella del segnale originale, significa che sono state rimosse troppe informazioni. In questo caso, l'algoritmo aumenta il valore di k, mantenendo più componenti principali e preservando quindi più informazioni. Questo processo continua fino a quando la potenza denoised non raggiunge o supera quella originale. Se k raggiunge un limite massimo, che corrisponde al 65% di $k_{\max}$, il ciclo si interrompe per evitare di preservare troppe componenti singolari che potrebbero contenere il rumore del segnale originale.

Al contrario quando la potenza del segnale denoised risulta essere superiore al 110% della potenza originale, l'algoritmo diminuisce il valore di k, rimuovendo alcune componenti principali. In questo caso, l'algoritmo riduce progressivamente k, e il segnale denoised viene ricostruito ad ogni iterazione, finché la potenza denoised non scende sotto il 110% della potenza originale. Anche in questo caso, il ciclo si interrompe quando k scende sotto il 30% di $k_{\max}$, per evitare di rimuovere troppe informazioni nel segnale.

Durante il processo, per ogni iterazione, il segnale denoised viene ricostruito con il nuovo valore di k, che viene aggiornato in base alla potenza del segnale.

4.3 Segnale denoised

4.3.1 Creazione della matrice audio denoised

```
1
2       U_com = U(:, 1:k);
3       S_com = S(1:k, 1:k);
4       V_com = V(:, 1:k);
5
6
7       audioMatrix_denoised = U_com * S_com * V_com';
```

Dopo aver effettuato la selezione del parametro K, procediamo ad estrarre le prime K colonne di U e V e la parte superiore KxK della matrice Σ. Effettuando il prodotto $U_{com} * S_{com} * V'_{com}$ otteniamo la matrice audio denoised.

4.3.2 Ricostruzione del segnale

```
1       % Ricostruzizone del segnale audio denoised con overlap-add
2       audio_denoised = zeros(length(audio), 1);
3       for i = 1:numFrames
4           startIdx = (i - 1) * hopSize + 1;
5           endIdx = min(startIdx + frameLength - 1, length(audio));
6           audio_denoised(startIdx:endIdx) = audio_denoised(startIdx:
                endIdx) + audioMatrix_denoised(:, i);
7       end
```

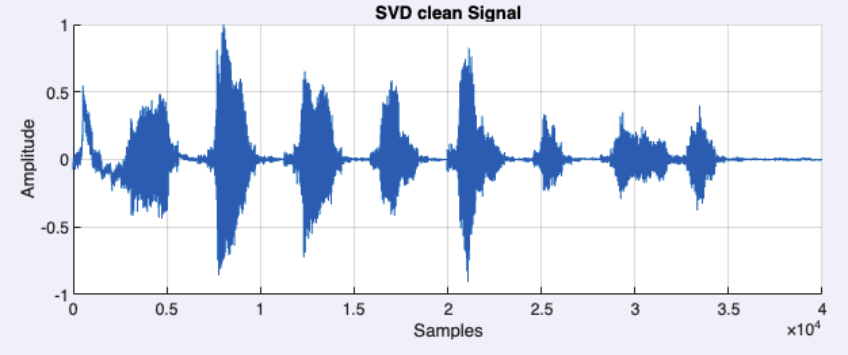
Per ricostruire il segnale dalla matrice denoised costruiamo un vettore vuoto delle stesse dimensioni del segnale iniziale, che andiamo a riempire all'interno del ciclo for. Per ogni finestra, che ricordiamo corrispondere ad una colonna della matrice audio denoised, viene calcolato l'indice d'inizio e fine e il vettore viene riempito con la colonna $i - esima$ della matrice audio denoised alla quale viene sommata la porzione del segnale già calcolata nella finestra precedente, poichè le finestre si sovrappongono e dunque includono anche porzioni delle finestre adiacenti. Questo approccio prende il nome di **overlap-add**.
Successivamente si procede **normalizzando** il vettore ottenuto:

```
1       audio_denoised = audio_denoised / max(abs(audio_denoised));
```

La normalizzazione è un passo molto importante poichè evita problemi di **clipping**, un fenomeno che si verifica quando un segnale supera il limite massimo di valore che può essere rappresentato in termini di ampiezza e causa distorsione nel segnale. Inoltre il clipping può anche causare **danni hardware** ad amplificatori o altoparlanti, poiché il segnale sovraccaricato può generare onde di alta potenza che danneggiano i componenti elettronici.
Solitamente il segnale denoised ottenuto presenta ampiezza molto elevata a causa del processo di overlap-add, poiché le finestre si sovrappongono, alcuni campioni del segnale denoised verranno sommati più volte, il che porta ad un aumento dell'ampiezza.

Infine plottiamo e ascoltiamo il segnale denoised ottenuto:

```
1    plot(app.SvdcleanSignalAxes,audio_denoised)
2    sound(audio_denoised, fs);
```



5 Analisi dei risultati

5.1 Andamento temporale

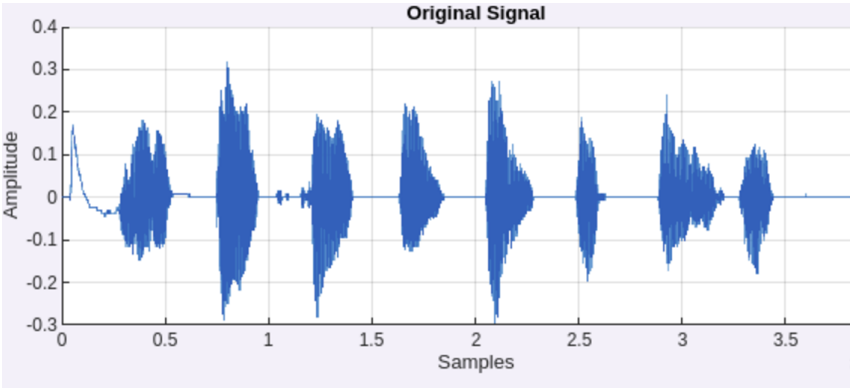


Figura 1: Segnale iniziale privo di rumore

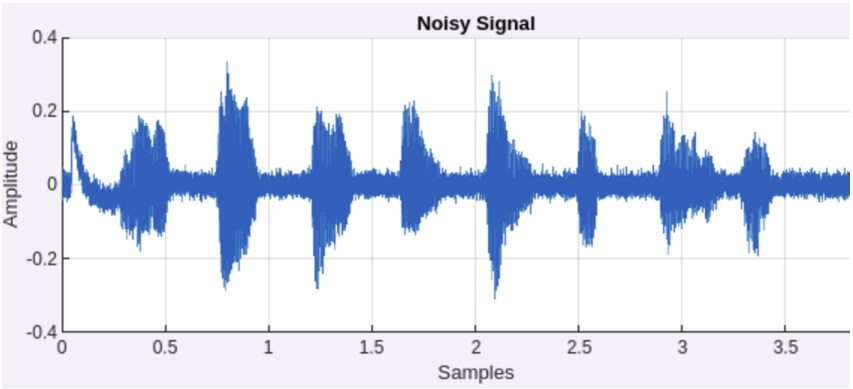


Figura 2: Segnale con l'aggiunta del rumore gaussiano

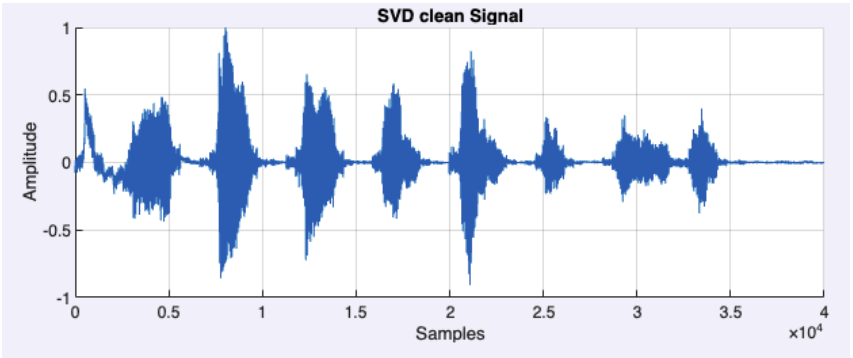


Figura 3: Segnale denoised ottenuto tramite SVD con scelta ottimale del parametro K

Si nota come il denoising ha ridotto significativamente il rumore cercando di preservare il contenuto del segnale. Il segnale ha un andamento simile a quello originale con piccole differenze dovute ad imprecisioni nel processo di denoising.

5.2 Analisi dello spettro di frequenza

```
N = length(audio);
N_denoised = length(audio_denoised);
if N_denoised > N
    N_denoised = N;
end

audio_fft = abs(trasformata_veloce(audio));
audio_denoised_fft = abs(trasformata_veloce(audio_denoised(1:
    N_denoised)));
```

Per effettuare lo studio dello spettro di frequenza procediamo prendendo i primi N campioni di entrambi i segnali, questo poichè il processo di ricostruzione del segnale audio denoised può produrre un segnale più lungo di quello originale a causa di sovrapposizioni ad esempio nel caso dell'overlap-add.
Successivamente si procede applicando la funzione della trasformata veloce, mostrata nella sezione dei fondamenti teorici, sul segnale originale e sul segnale denoised, cosi da poter lavorare nel dominio delle frequenze.
La funzione **abs** calcola il modulo di ogni valore della trasformata, poiché quest'ultima ci restituisce segnali complessi; questo modulo prende il nome di **magnitudine** e rappresenta l'intensità delle frequenze nell'audio.
Per poter visualizzare lo spettro delle frequenze, procediamo creando l'asse delle frequenze:

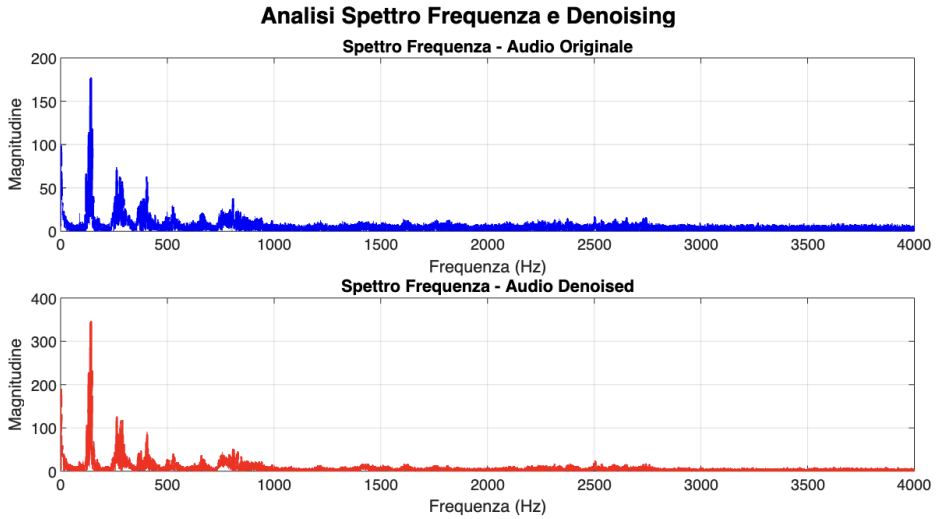
```
f = (0:length(audio_fft)-1) * fs / length(audio_fft); %
    Frequenza corrispondente per ogni punto FFT
```

L'espressione **(0:length(audio_fft)-1) * fs** genera un vettore contenente le frequenze associate ai valori delle trasformate, in particolare **0:length(audio_fft)-1** crea un vettore contenente i valori interi da 0 al numero dei valori delle trasformate-1 e moltiplicare per la frequenza di campionamento fs ci consente di ottenere le frequenze reali. Infine dividendo per la lunghezza del vettore delle trasformate normalizziamo la scala delle frequenze in modo che siano distribuite correttamente tra 0 e $\frac{fs}{2}$. Ci serve che le frequenze siano distribuite correttamente in questo intervallo poiché $\frac{fs}{2}$ rappresenta la **frequenza di Nyquist** ed è la massima frequenza rappresentabile senza **aliasing**, un fenomeno che si verifica quando le alte frequenze si sovrappongono creando distorsioni.

Ora procediamo limitando la parte di spettro che andiamo ad analizzare, eliminando le frequenze negative che per segnali reali rappresentano una semplice riflessione simmetrica del contenuto spettrale delle frequenze positive:

```
% Limitiamo le frequenze per la meta' (da 0 a Nyquist)
f = f(1:floor(length(f)/2));
audio_fft = audio_fft(1:floor(length(audio_fft)/2));
audio_denoised_fft = audio_denoised_fft(1:floor(length(
    audio_denoised_fft)/2));
```

Plottando lo spettro di magnitudine dell'audio col rumore e dell'audio denoised otteniamo i seguenti grafici:

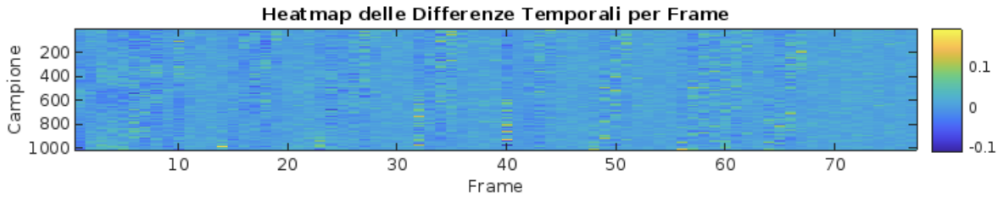


Nel primo grafico si nota come il rumore gaussiano ha una distribuzione uniforme nel dominio delle frequenze contribuendo in modo simile a tutte le bande. I picchi più evidenti soprattutto a basse frequenze rappresentano la parte utile del segnale ovvero il parlato. Nel secondo grafico si osserva come la parte rumorosa che si nota di più nelle alte frequenze, sia stata attenuata.

5.3 Heatmap delle differenze temporali per frame

```
diff_matrix = abs(audioMatrix) - abs(audioMatrix_denoised);
imagesc(diff_matrix);
colorbar;
title('Heatmap delle Differenze di Frequenza nel Tempo');
xlabel('Frame');
ylabel('Campione');
```

Procediamo calcolando la matrice di differenza assoluta tra la matrice del segnale col rumore e quella del segnale denoised. Otteniamo cosi il seguente grafico:



Ogni punto del grafico rappresenta quanto un campione è stato modificato dal processo di denoising. Le zone scure sono le zone a bassa differenza ovvero in cui il segnale denoised è molto simile a quello rumoroso, indicando che questi campioni rappresentano porzioni di segnale utile. Le zone chiare invece sono le zone ad alta differenza in cui il segnale denoised differisce molto da quello rumoroso, indicando delle zone in cui il segnale conteneva molto rumore.

6 Bibliografia

- Materiale didattico del corso "Metodi numerici avanzati" della professoressa Elisa Francomano
- Matrix computations, Golub and Van Loan
- Documentazione MATLAB
- Numerical linear algebra, Trefethen and Bau
- Lezioni di Teoria dei Segnali, Giovanni Garbo, Giovanni Mamola e Stefano Mangione