Riduzione del rumore in un segnale audio tramite SVD

Gabriele Baiamonte e Riccardo Rubino

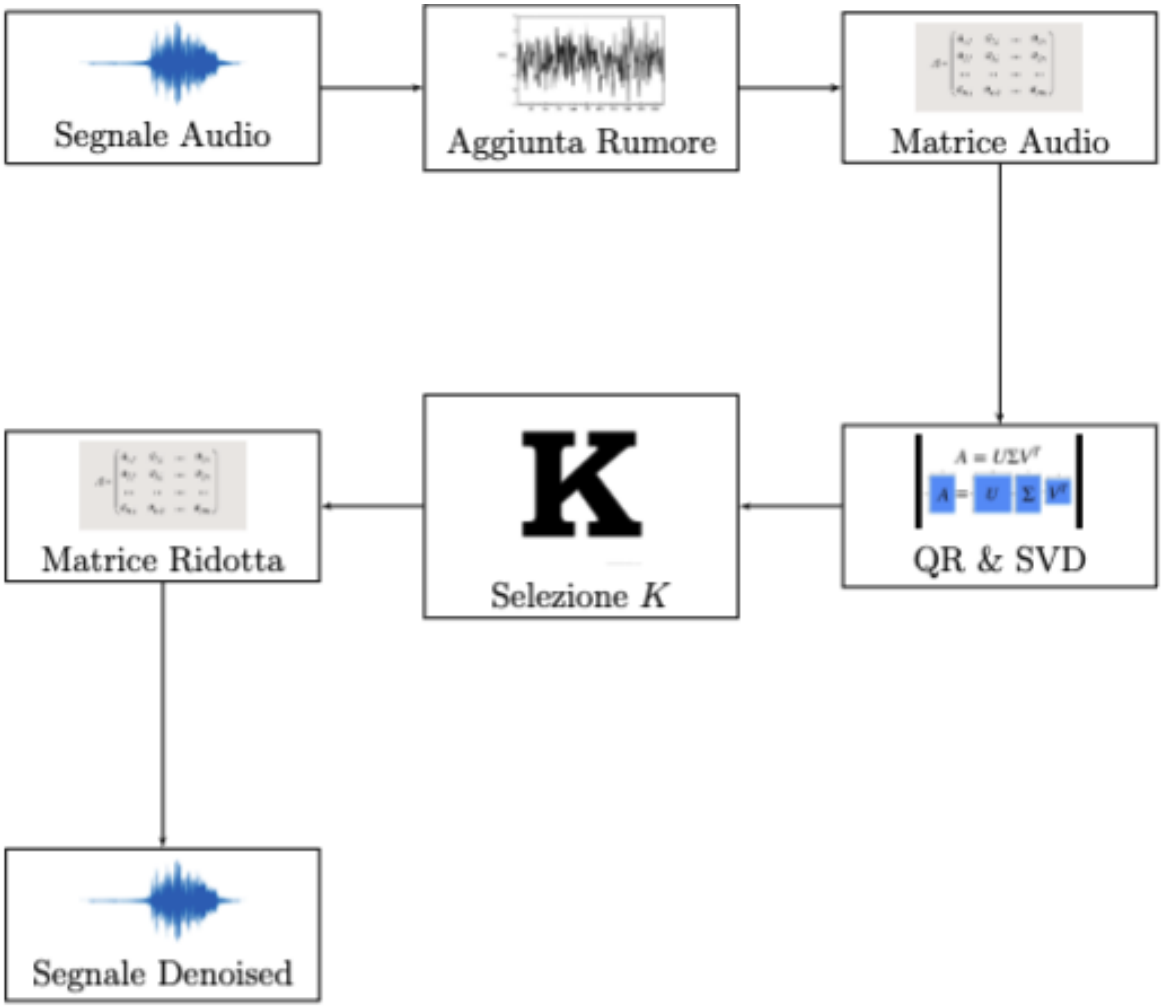
Università degli Studi di Palermo

Indice

- Introduzione al progetto
- Cos'è l'SVD
- Algoritmo per la riduzione del rumore nel segnale audio e richiami teorici
- Risultati ottenuti

Introduzione al progetto

Schema a blocchi



Cos'è l'SVD?

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.

$$A = U\Sigma V^T \tag{1}$$

Dove $U \in R^{mxm}$ e $V \in R^{nxn}$ sono due matrici ortogonali, in particolare:

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:

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

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 $A^T A$.
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 esistenza SVD

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.

Dimostrazione esistenza SVD

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

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

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:

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

$$= (\sigma_1^2 + w^T w)^2 + \|Bw\|_2^2 \;\geq\; (\sigma_1^2 + w^T w)^2.$$

Dimostrazione esistenza SVD

Da cui segue che:

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

Ma $\|A\|_2^2 = \sigma_1^2$ e $\|A\|_2 = \|A_1\|_2$ poiché A_1 si ottiene tramite trasformazioni ortogonali di A. Allora:

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

Quindi:

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

Ne consegue:

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

Notiamo che abbiamo dimostrato il caso base dell'induzione dato che per $n = 1$ o $m = 1$ abbiamo ottenuto la decomposizione SVD.

Dimostrazione esistenza SVD

Dunque supponiamo che il teorema sia vero per una matrice $(m-1) \times (n-1)$. Dimostriamo ora che è vero per una matrice generica $(A \in \mathbb{R}^{m \times n})$.
Per ipotesi induttiva B avrà una decomposizione SVD del tipo

$$B = U_2 \, \Sigma_2 \, V_2^T,$$

da cui:

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

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

Dimostrazione esistenza SVD

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 (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.

SVD - Come sono composte le matrici U, Σ e V?

Consideriamo il prodotto $A^T A$:
$A^T A = (V\Sigma^T U^T)(U\Sigma V^T) = V\Sigma^T \Sigma V^T$
Che possiamo riscrivere come:
$(A^T A)V = V(\Sigma^T \Sigma)$
Cioè 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.

SVD - Come sono composte le matrici U, Σ e V?

$\Sigma^T\Sigma$ è una matrice di dimensioni $max(n,m)xmax(n,m)$ contente sulla diagonale il quadrato dei valori singolari. Infatti si può dimostrare facilmente nel caso in cui Σ sia di dimensione $4x6$:

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

SVD - Come sono composte le matrici U, Σ e V?

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

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

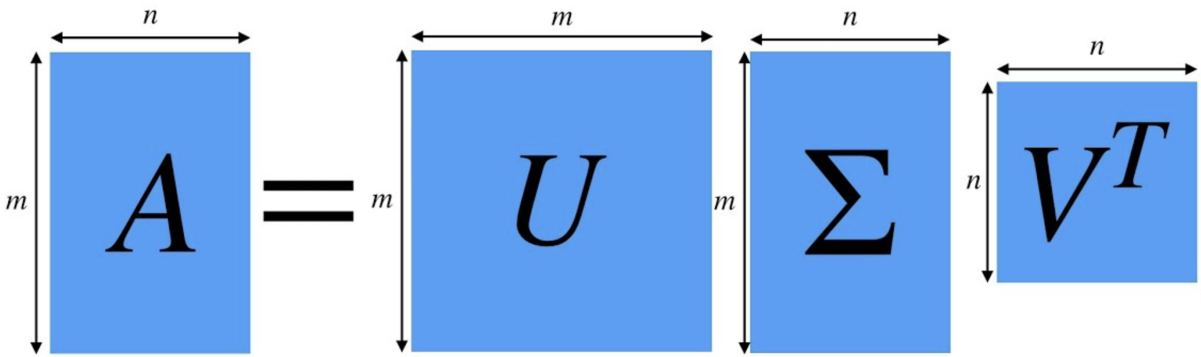
Possiamo quindi affermare che gli elementi di D corrispondono agli autovalori di $A^T A$:

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

SVD - Come sono composte le matrici U, Σ e V?

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

SVD



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.

SVD - 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.

SVD - 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.

Algoritmo per la riduzione del rumore

Caricamento del file audio e inserimento del rumore (1/3)

```
[x,fs] = audioread('voice.wav');
if size(audio, 2) == 2
    audio = mean(audio, 2);
end
sound(x,fs);
plot(app.OriginalSignalAxes, x);
```

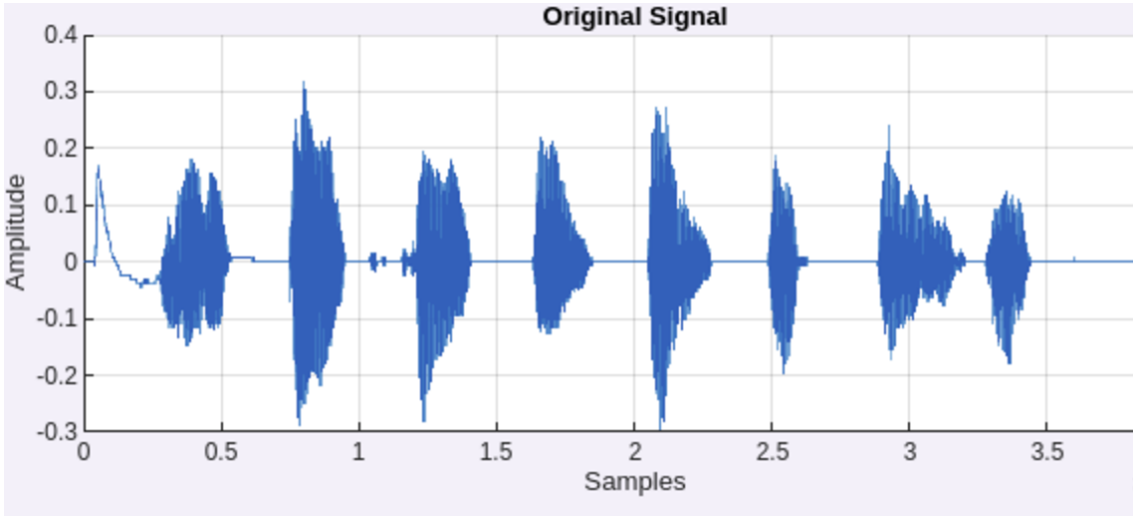
Il file audio viene caricato tramite **audioread**, che crea un vettore **x**. Se l'audio è stereo, lo convertiamo in mono (media dei due canali). **sound** ci permette di ascoltare il segnale. **plot(app.OriginalSignalAxes,x)** mostra la rappresentazione temporale del segnale.

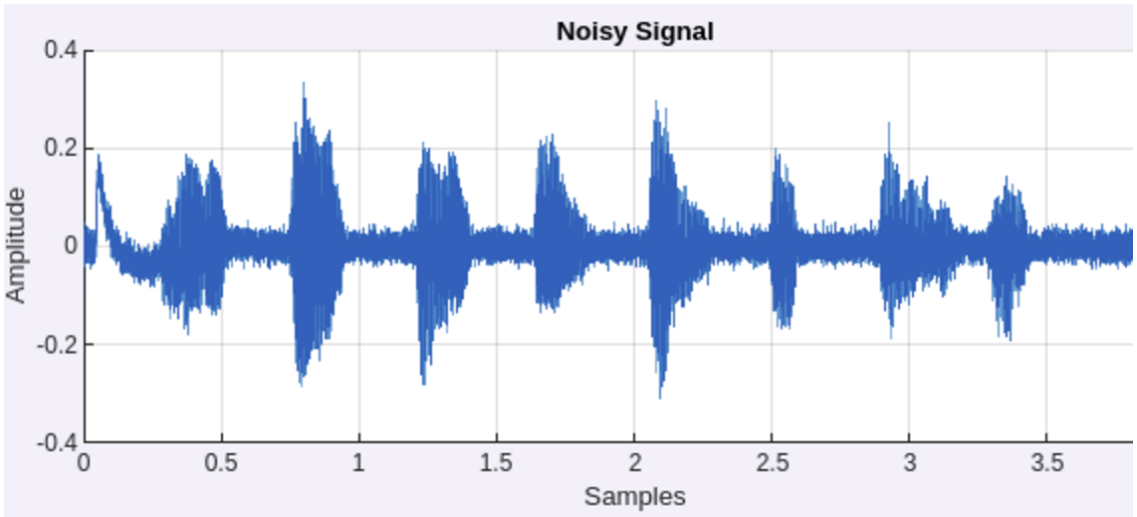
Caricamento del file audio e inserimento del rumore (2/3)

```
audio = awgn(x,10,'measured');
filename = 'noisyvoice.wav';
audiowrite(filename, audio, fs)
pause(5)
sound(audio,fs);
plot(app.NoisySignalAxes, audio);
```

- **awgn(x,10,'measured')**: aggiunge al segnale **x** del rumore bianco gaussiano (SNR=10 dB), stimando l'energia del segnale.
- Salviamo il nuovo segnale rumoroso (`noisyvoice.wav`) e lo plottiamo per vederne la forma d'onda.
- Questo rumore gaussiano tende a distribuire la sua energia in tutte le componenti della matrice, ma è particolarmente influente sulle componenti a basso valore singolare da cui l'efficacia della SVD nel denoising.

Caricamento del file audio e inserimento del rumore (3/3)





Suddivisione in finestre (1/6)

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 esattamente dove finisce la precedente.
- La sovrapposizione cattura meglio le transizioni (cambi di tono o timbro).
- Si ottiene una transizione più fluida tra i vari frammenti.

Suddivisione in finestre (2/6)

Parametri della finestra:

- **frameLength** = 1024 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 Rappresenta la distanza tra l'inizio di una finestra e l'inizio della finestra successiva.

Osservazione: Nella matrice finale, ogni colonna corrisponderà a una finestra.

Suddivisione in finestre (3/6)

Calcolo del numero di finestre:

$$\texttt{numFrames} = \left\lfloor \frac{\text{length}(\texttt{audio}) - \texttt{overlap}}{\texttt{hopSize}} \right\rfloor$$

- length(audio) - overlap è il numero di campioni effettivamente utilizzabili dopo aver considerato l'`overlap`.
- Dividendo per `hopSize` otteniamo quante finestre possiamo ricavare.
- `floor` arrotonda il risultato all'intero più vicino (verso il basso).

Motivazione: non possiamo avere un numero non intero di finestre.

Suddivisione in finestre (4/6)

Costruzione della matrice

Ogni colonna è una finestra da 1024 campioni.

```
startIdx = (i - 1) * hopSize + 1;
endIdx   = startIdx + frameLength - 1;
audioMatrix(:, i) = audio(startIdx:endIdx);
```

- `startIdx` indica l'indice di partenza della finestra *i*.
- `endIdx` indica l'indice finale della finestra *i*.
- `audioMatrix(:, i)` contiene i campioni della finestra *i* estratti dal segnale.

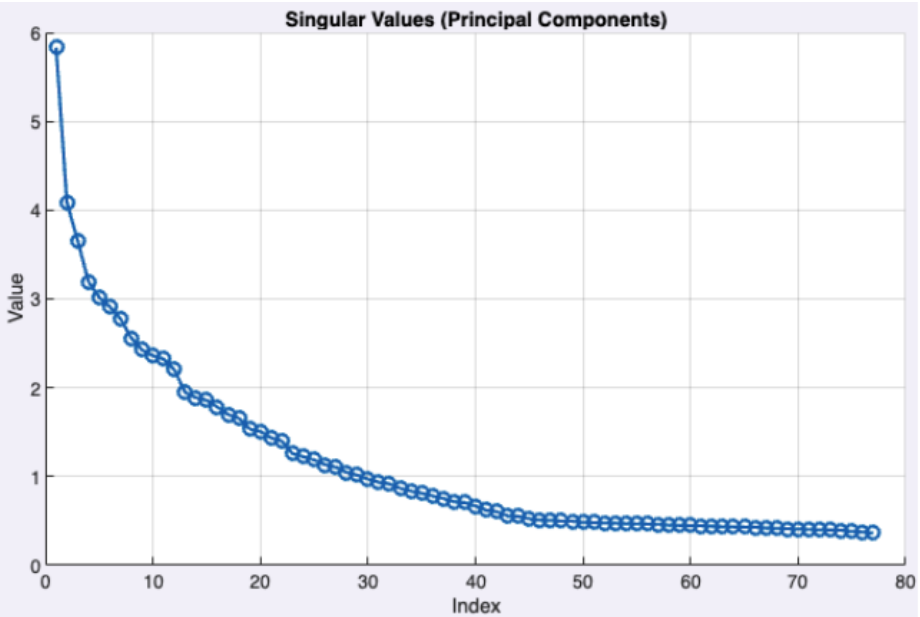
Codice completo per la creazione della matrice

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

% Crea la matrice audio con finestre sovrapposte
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
```

Applicazione SVD sulla matrice audio(Codice Principale)

```
% Applicazione della SVD alla matrice audio
[U, S, V] = svd_qr_hessenberg_shift(audioMatrix);
% Estrazione dei valori singoli
singularValues = diag(S);
plot(app.Valorisingolari,singularValues, 'o-', 'LineWidth',
1.5, 'MarkerSize', 6);
```



Ma come è fatta la funzione svd_qr_hessenberg_shift?

```
    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
```

Ma come è fatta la funzione svd_qr_hessenberg_shift?

- **[eigvals, eigvecs] = qr_hessenberg_shift(AtA):** Attraverso la funzione 'qr_hessenberg_shift' vengono calcolati autovettori e autovalori della matrice $A^T A$.
- **singular_values = sqrt(abs(eigvals)):** Si calcolano i valori singolari applicando la radice quadrata agli autovalori.
- **[sorted_singular_values, idx] = sort(singular_values, 'descend'):** È necessario ordinare i valori singolari in ordine decrescente.

Ma come è fatta la funzione svd_qr_hessenberg_shift? (continua)

- **V = eigvecs(:, idx):** Si ordinano gli autovettori utilizzando l'indice di ordinamento 'idx'.
- **S = diag(sorted_singular_values):** Costruiamo la matrice S, che è diagonale e contiene i valori singolari.
- **U = A * V / S:** Usando $U = AVS^{-1}$, calcoliamo U. Poiché S è diagonale, l'inverso corrisponde all'inverso dei suoi elementi sulla diagonale.
- **Normalizzazione di U:**

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

Alla fine normalizziamo le colonne di U.

qr_hessenberg_shift: Idea di base

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 Hessenberg.

qr_hessenberg_shift: Preparazione

```
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);
```

qr_hessenberg_shift: Ciclo principale

```
for iter = 1:max_iterations
    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
    mu = wilkinson_shift(H(max(1,m-1):m, max(1,m-1):m));
    [Q, R] = qrgivensnostra(H - mu * eye(n));
    H_new = R * Q + mu * eye(n);
    V = V * Q;
    if norm(H - H_new, 'inf') < tol
        break;
    end
    H = H_new;
end
```

qr_hessenberg_shift: Autovalori e autovettori

```
% 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
```

Analisi del codice

questa funzione applica il metodo QR per il calcolo di autovalori e autovettori, inizialmente:

```
    max_iterations = 10000;
    tol = 1e-5;

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

     % Inizializzazione
    n = size(H, 1);
    V = eye(n);
```

Applichiamo la riduzione in forma di Hessenberg alla matrice A, definiamo anche una matrice V che servirà per accumulare gli autovettori durante le iterazione, vedremo meglio questo concetto in seguito.

Analisi del codice. Riduzione in forma di hessemberg

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 sottodiagonale, e una matrice di Hessenberg inferiore li ha sopra la prima sovradiagonale. 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, può avvenire attraverso le trasformazioni di Householder, in particolare eliminano i termini al di sotto della prima sottodiagonale.

Analisi del codice

```
   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
```

Questo blocco analizza gli elementi della sottodiagonale di H, se in base alla tolleranza un elemento è molto vicino a 0 allora si restringe l'attenzione al blocco più piccolo di H.

Analisi del codice. Shift di Wilkinson

```
mu = wilkinson_shift(H(max(1,m-1):m, max(1,m-1):m));
```

Questo blocco applica lo shift di Wilkinson, che permette di migliorare la convergenza del metodo QR.

Analisi del codice. 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:

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

Analisi del codice. Shift di Wilkinson:Implementazione

```
function mu = wilkinson_shift(submatrix)
    if size(submatrix, 1) == 1
        mu = submatrix;
        return;
    end
    if size(submatrix, 1) == 2
        a = submatrix(1,1);
        b = submatrix(1,2);
        c = submatrix(2,1);
        d = submatrix(2,2);
        disc = sqrt((a + d)^2 - 4 * (a*d - b*c));
        % Selezione dello shift
        if abs(a + d + disc) > abs(a + d - disc)
            mu = a + d - disc;
        else
            mu = a + d + disc;
        end
    else
        % Per matrici più grandi, usa l'ultimo elemento come shift
        mu = submatrix(end, end);
    end
end
```

Analisi del codice

```
[Q, R] = qrgivensnostra(H - mu * eye(n));
H_new = R * Q + mu * eye(n);
V = V * Q;
```

Ora applichiamo la fattorizzazione QR attraverso la funzione qrgivensnostra che applica la fattorizzazione QR usando le rotazioni di Givens, successivamente aggiorniamo la matrice H e V.

Analisi del codice. Funzione qrgivensnostra

```
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
```

Funzione ausiliaria: calcolo_parametri_givens_progetto

```
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
```

Analisi del codice. fattorizzazione QR usando le rotazioni 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:

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

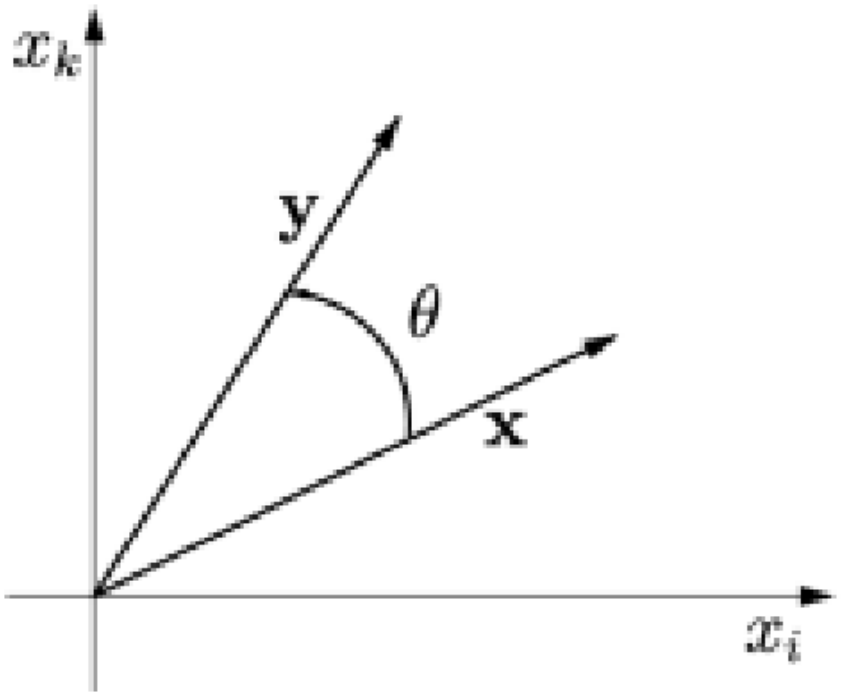
Analisi del codice. fattorizzazione QR usando le rotazioni di Givens

in forma esplicita:

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

Analisi del codice. fattorizzazione QR usando le rotazioni di Givens

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).



Analisi del codice. fattorizzazione QR usando le rotazioni di Givens

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:

$$\cos\theta = \frac{a_{ii}}{\sqrt{a_{ii}^2 + a_{ij}^2}}$$

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

Analisi del codice. fattorizzazione QR usando le rotazioni di Givens

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.

Analisi del codice

```
    if norm(H - H_new, 'inf') < tol
     break;
end
H = H_new;

eigenvalues = diag(H);
```

La norma infinito controlla la differenza tra H e H_new, se la differenza è sotto la soglia l'algoritmo termina. Gli autovalori di A sono approssimati come gli elementi diagonali di H.

Analisi del codice

```
   eigenvectors = P * V;
for i = 1:n
    eigenvectors(:,i) = eigenvectors(:,i) / norm(eigenvectors(:,i))
end
```

Gli autovettori vengono calcolati utilizzando la matrice di trasformazione P e la matrice V, che contiene gli autovettori accumulati durante il processo.

Analisi del codice. Calcolo degli autovettori

All'inizio dell'implementazione avevamo definito 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.

Analisi del codice. Calcolo degli autovettori

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

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

Applicazione SVD sulla matrice audio(Codice Principale)

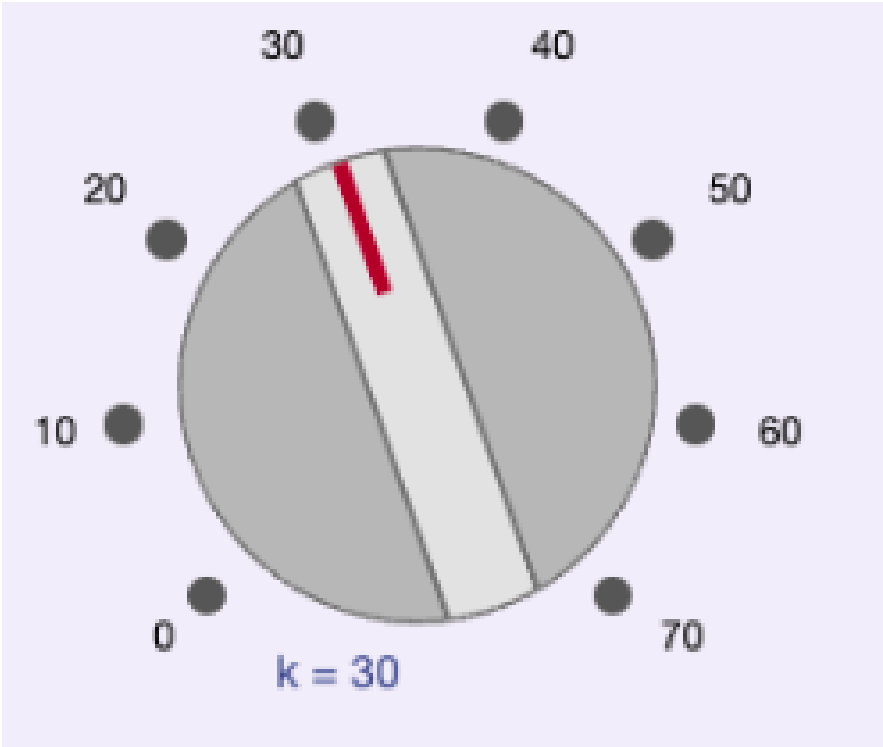
```
% Applicazione della SVD alla matrice audio
[U, S, V] = svd_qr_hessenberg_shift(audioMatrix);
% Estrazione dei valori singoli
singularValues = diag(S);
plot(app.Valorisingolari,singularValues, 'o-', 'LineWidth',
1.5, 'MarkerSize', 6);
```

A questo punto dopo aver visto nel dettaglio la funzione svd_qr_hessenberg_shift e le rispettive funzioni interna qeust'ultima, il codice continua effettuando il plot dei valori singolari che si trovano sulla diagonale della matrice S.

Scelta del parametro k(Codice Principale)

Abbiamo deciso di dornire due possibilita per la selezione del k:

- manualmente atttraverso l'interfaccia grafica.



- attraverso un algoritmo che utilizza le informazioni contenute nei valori singolari e la potenza del segnale sceglie un valore di k ottimale.

Scelta automatica del parametro k: Calcolo della potenza del segnale originale

- La potenza del segnale originale è calcolata con la formula:

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

- Implementazione del calcolo in codice:

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

- Permette di stabilire un riferimento per il confronto con il segnale denoised.

Scelta automatica del parametro k: Selezione iniziale di k

- Scelta di k per mantenere almeno il 50% delle informazioni.

```
% Fissa k al 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
```

Scelta automatica del parametro k: Calcolo della potenza del segnale denoised

- Ricostruzione del segnale denoised con il valore iniziale di k e calcolo della potenza del segnale ricostruito con il 50% delle informazioni

```
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);
```

Scelta automatica del parametro k: Ottimizzazione del valore di k

- Regola k in base alla potenza del segnale denoised:

```
if denoisedPower < originalPower
    while denoisedPower < originalPower && k < k_max
        % Incrementa k
    end
elseif denoisedPower > 1.1 * originalPower
    while denoisedPower > 1.1 * originalPower && k > ceil(0.3 * k_m
        % Decrementa k
    end
end
```

- Mantiene un equilibrio tra informazioni preservate e riduzione del rumore.

Scelta automatica del parametro k:Obiettivo raggiunto.

- L'algoritmo ottimizza il valore di k per bilanciare:
  - La quantità di informazioni preservate nel segnale.
  - La potenza del segnale denoised rispetto a quella originale.
- **Aumento di k:**
  - Se la potenza del segnale denoised è inferiore a quella originale, k viene incrementato.
  - Il processo si arresta al 65% di $k_{\max}$ per evitare di includere rumore.
- **Riduzione di k:**
  - Se la potenza del segnale denoised supera il 110% di quella originale, k viene ridotto.
  - Il processo si arresta al 30% di $k_{\max}$ per evitare di rimuovere troppe informazioni.
- Ad ogni iterazione, il segnale denoised viene ricostruito con il valore aggiornato di k.

Ricostruzione del segnale denoised

- Dopo la selezione del parametro k, estraiamo le prime k colonne di U e V, e la porzione $k \times k$ di Σ.
- La matrice audio denoised viene calcolata come:

$$\textbf{audioMatrix\_denoised} = U_{com} \cdot S_{com} \cdot V_{com}^{T}$$

```
% Estrazione le prime k componenti
U_com = U(:, 1:k);
S_com = S(1:k, 1:k);
V_com = V(:, 1:k);

% Ricostruzione della matrice audio ridotta
audioMatrix_denoised = U_com * S_com * V_com';
```

Ricostruzione del segnale denoised:

- Il segnale audio denoised viene ricostruito utilizzando il metodo **overlap-add**.
- Per ogni finestra, si calcolano l'indice di inizio e fine, e si aggiorna il vettore del segnale sovrapponendo le finestre.

```
% Ricostruzione del segnale audio denoised
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
```

Ricostruzione del segnale denoised: normalizzazione del segnale

- Il segnale ricostruito viene **normalizzato** per evitare problemi di **clipping**.
- La normalizzazione riduce l'ampiezza del segnale per mantenere i valori entro il limite massimo.

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

- Questo evita distorsioni nel segnale e previene possibili **danni hardware**.

Risultati ottenuti

Analisi dei risultati: Andamento temporale (1/2)

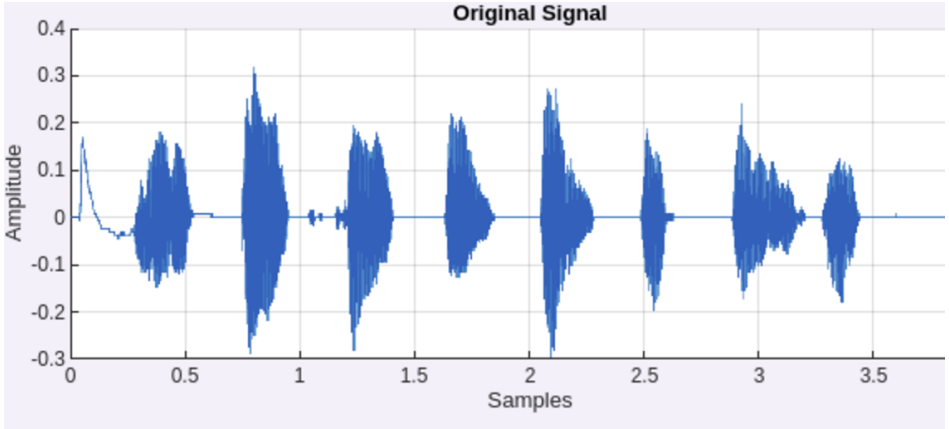


Figure: Segnale iniziale privo di rumore

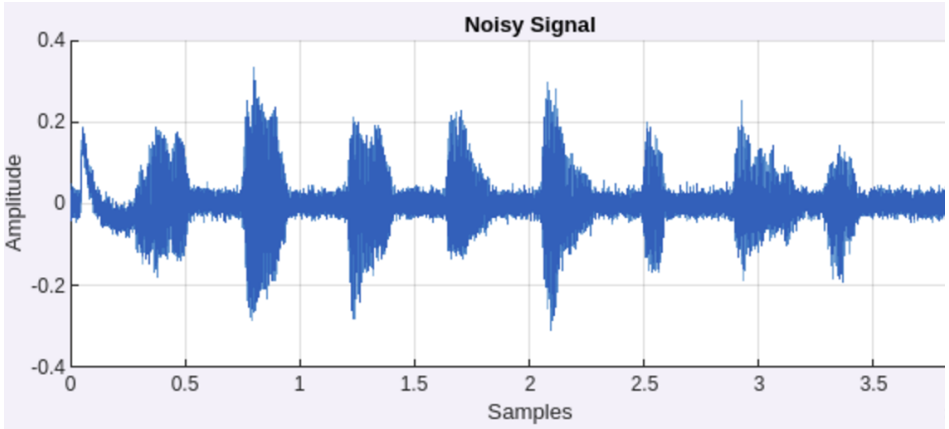


Figure: Segnale con l'aggiunta del rumore gaussiano

Analisi dei risultati: Andamento temporale (2/2)

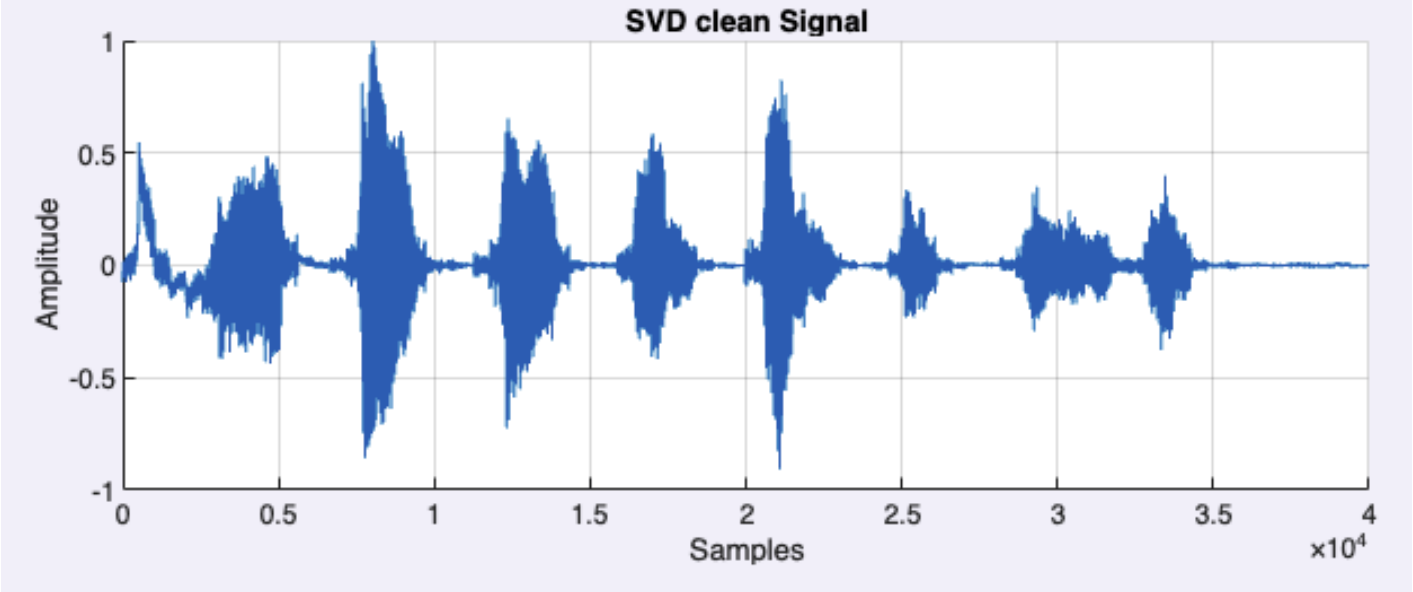


Figure: Segnale denoised ottenuto tramite SVD con scelta ottimale del parametro K

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

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 lo **studio dello spettro di frequenza**:

- Si considerano i primi N campioni di entrambi i segnali (l'audio denoised può risultare più lungo per via del processo di ricostruzione, ad esempio overlap-add).
- Viene applicata la **trasformata veloce** (FFT) a entrambi i segnali, ricavandone il modulo (abs), la cosiddetta *magnitudine* che rappresenta l'intensità delle frequenze nell'audio.

Creazione asse delle frequenze

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

- `(0:length(audio_fft)-1)` genera l'insieme di indici tra 0 e `length(audio_fft)-1`.
- Moltiplicare per `fs` converte gli indici nelle corrispettive frequenze in Hz.
- Dividere per `length(audio_fft)` normalizza la scala delle frequenze, distribuite tra 0 e $fs/2$.

Frequenza di Nyquist:

$$\frac{fs}{2}$$

rappresenta la massima frequenza campionabile senza incorrere in aliasing.

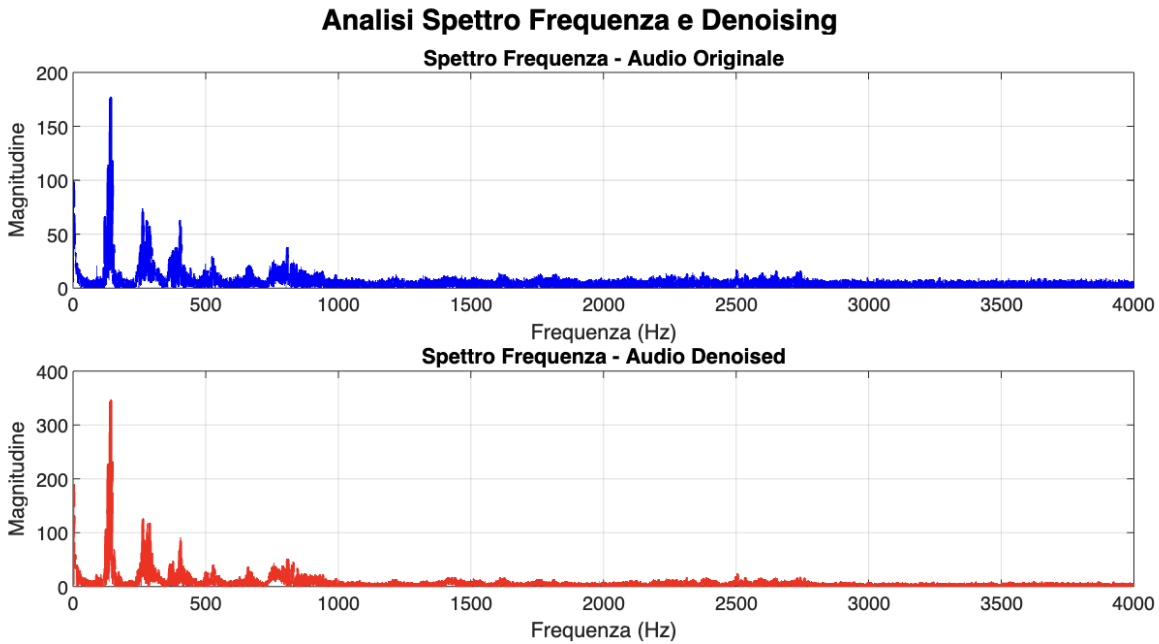
Limitazione alle frequenze positive

```
% Limitiamo le frequenze a [0, 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))
```

- I segnali reali hanno uno spettro *simmetrico*, perciò la parte "negativa" è ridondante.
- Con `f(1 : floor(length(f)/2))` prendiamo soltanto la banda [0, Nyquist].

Grafici ottenuti

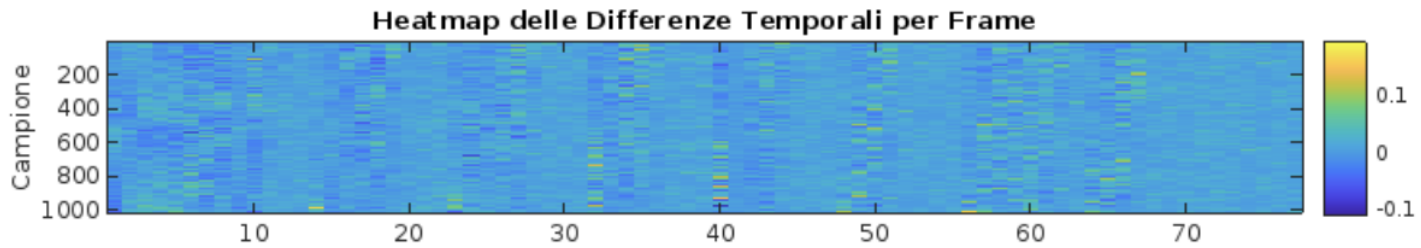
Il *rumore gaussiano* è distribuito in modo uniforme sullo spettro, mentre il segnale utile (parlato) mostra picchi concentrati, soprattutto alle basse frequenze.



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');
```

- `diff_matrix` rappresenta la differenza assoluta fra i campioni di `audioMatrix` (con rumore) e `audioMatrix_denoised`.
- Le zone scure (*bassa differenza*) indicano campioni rimasti quasi invariati, ossia porzioni di segnale utile.
- Le zone chiare (*alta differenza*) mostrano i punti in cui il denoising ha operato una riduzione significativa del rumore.



Analisi uditiva

Bibliografia

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