UNIVERSITÀ DEGLI STUDI
DI PERUGIA

Tesina di Signal processing and optimization for Big Data

Corso di Laurea Magistrale in Ingegneria Informatica e Robotica

Curriculum Data Science

A.A. 2022/23

Docente **Paolo Banelli**

Compressed sensing algorithms

350395 Filippo Francisci email: filippo.francisci@studenti.unipg.it

SOMMARIO

Introduzione

L'obiettivo di questo progetto è quello di confrontare le prestazioni di diversi algoritmi di *compressed sensing*, per stabilire quanto bene riescono a ricostruire un vettore sparso, confrontandone parametri prestazionali di interesse.

Descrizione del problema

Il problema del *compressed sensing* si basa sulla possibilità di poter ricostruire un segnale sparso da *M* osservazioni, noto che $M \leq N$, definendo una matrice ϕ (matrice di compressione) rispetto ad un segnale *x*, il quale ammette una certa rappresentazione sparsa in un qualche dominio, definita dalla combinazione con la matrice ψ ortonormale. Tale descrizione è riassunta nella seguente formulazione:

$$\underline{y} = \phi * \psi * \underline{s} = \theta * \underline{s}, \qquad \text{con } \theta \text{ pari a } \phi * \psi.$$

La ricostruzione del vettore sparso *s* è possibile tramite la risoluzione di un sistema indeterminato (il numero di misurazioni *M* è inferiore alla lunghezza *N* del vettore sparso) e si ottiene solo se, a partire da diversi vettori sparsi, ottengo anche diverse misurazioni *y*:

$$\theta * \underline{s}_1 \neq \theta * \underline{s}_2 \implies \theta * (\underline{s}_1 - \underline{s}_2) \neq 0$$

Se s_1 ed s_2 sono due vettori sparsi con sparsità *K*, il vettore $z = s_1 - s_2$ sarà al più *2K* sparso (quando tutti i coefficienti diversi da *0* si trovano in posizioni differenti).
Quindi tutti i possibili insiemi di *2K* colonne devono essere linearmente indipendenti.
Si definisce con *spark* di θ il minimo numero di colonne linearmente dipendenti.
Se vale che $\| \underline{s}^* \|_0 \leq \frac{1}{2} spark(\theta)$, allora $\underline{s}^*$ è la soluzione ottima del problema seguente che individua la soluzione più sparsa possibile (minimizzazione del numero di coefficienti non nulli):

$$\text{argmin} \| \underline{s}^* \|_0 \qquad \text{s.t} \quad \theta * \underline{s} = \underline{y}$$

Se si vuole invece individuare la soluzione più *K-sparsa*, allora si richiede che:

$$spark(\theta) \geq 2K + 1$$

Assumendo la matrice θ a rango pieno (pari a *M*), lo $spark(\theta)$ sarà *M + 1*, da cui la condizione $M \geq 2K$. Basta un numero di osservazioni pari ad almeno *2K* per ricostruire correttamente il segnale sparso, ma trovare il minimo numero di colonne linearmente dipendenti ha complessità combinatoria e inoltre il problema di ricostruzione della rappresentazione sparsa norma *0* (che conta il numero di componenti del vettore diverse da *0*) è *NP-Hard*, per cui va rilassato per poterlo risolvere. Per semplificare la ricerca della soluzione si passa per altra via. Se θ viene costruita con campioni *i.i.d* randomici, la probabilità che lo *spark* di θ sia maggiore uguale a *2K + 1* è all'incirca pari a *1*. La soluzione è quindi quella di ricercare matrici con

colonne più ortogonali possibili (più incoerenti tra loro). Definito $\mu(\theta)$ come il coefficiente di mutua coerenza (calcola il coefficiente di correlazione facendone poi il massimo), che descrive quanto le colonne di θ sono ortogonali tra loro (in modo che l'informazione su ogni componente non nulla di *s* sia indipendente rispetto a quella degli altri), si riesce ad identificare una relazione di interesse:

$$M \geq 1 + \frac{1}{\mu(\theta)}$$

Inoltre, $\mu(\theta)$ è definito da un *lower bound* tale per cui $\mu(\theta) \geq \frac{1}{\sqrt{M}}$, per $N \gg M$, che va ad identificare la seguente relazione:

$$spark(\theta) \geq 1 + \sqrt{M} \geq 1 + \frac{1}{\mu(\theta)}$$

Essendo che $spark(\theta) \geq 2K + 1$, allora:

$$\sqrt{M} \geq 2K \Longrightarrow M \geq 4K^2$$

Il coefficiente di mutua coerenza è molto semplice da calcolare, ma il prezzo da pagare è che servono molte più osservazioni per ricostruire il segnale.

Una proprietà fondamentale è la *Restricted Isometry Property (RIP)* che viene espressa dalla seguente relazione:

$$(1 - \delta_k) \parallel \underline{s} \parallel_2^2 \leq \parallel \theta \underline{s} \parallel_2^2 \leq (1 + \delta_k) \parallel \underline{s} \parallel_2^2$$

In particolare, per δ_k tendente a *0* la norma euclidea tende ad essere preservata a seguito della moltiplicazione con la matrice θ, che in questo caso risulta essere ortonormale. Se la matrice θ rispetta la proprietà *RIP* rispetto ad un livello di sparsità pari a *2K*, ne consegue che per δ_{2k} tendente a *0*, la norma della differenza tra i due vettori sparsi viene preservata.

$$(1 - \delta_{2k}) \parallel \underline{s}_1 - \underline{s}_2 \parallel_2^2 \leq \parallel \theta \, (\underline{s}_1 - \underline{s}_2) \parallel_2^2 \leq (1 + \delta_{2k}) \parallel \underline{s}_1 - \underline{s}_2 \parallel_2^2$$

Questa proprietà garantisce la ricostruibilità del segnale anche quando soggetto a rumore. Se vale la proprietà *RIP* di livello *2K*, allora la ricostruzione del segnale avviene attraverso la risoluzione del problema già precedentemente definito, anche in presenza di rumore:

$$argmin \parallel \underline{s}^* \parallel_0 \qquad \text{s.t} \quad \theta \underline{s} = \underline{y}$$

Essendo, come già detto, un problema *NP-Hard*, si definisce un rilassamento dello stesso nelle seguenti possibili forme di risoluzione:

a) Basis Pursuit (l1 relaxation)

$$\text{argmin}_{\underline{s}} \parallel \underline{s} \parallel_1 \qquad s.t. \ \theta \underline{s} = \underline{y}$$

b) Quadratically costrained (Basis Pursuit)

$$\text{argmin}_{\underline{s}} \parallel \underline{s} \parallel_1 \qquad s.t. \ \parallel \theta \underline{s} - \underline{y} \parallel_2^2 \leq n$$

c) Lasso

$$\text{argmin}_{\underline{s}} \parallel \theta \underline{s} - \underline{y} \parallel_2^2 \qquad s.t. \quad \parallel \underline{s} \parallel_1 \leq \tau$$

La soluzione a norma *0* coincide con quella a norma *1* nel caso in cui vale la seguente relazione:

$$\mu(\theta) < \frac{1}{2K-1}$$

Inoltre, la ricostruzione del vettore sparso è ben condizionata nel caso in cui si sceglie una matrice θ in uno dei tre possibili seguenti modi:

- Campionamento uniforme e randomico da ipersfera a norma unitaria;
- Campionamento uniforme e randomico da distribuzione gaussiana a media nulla e varianza $\frac{1}{M}$;
- Campionamento da distribuzione simmetrica bernoulliana che assume valore $\pm\frac{1}{\sqrt{N}}$ con probabilità $\frac{1}{2}$;

Sotto queste condizioni, la proprietà *RIP* è rispettata per la matrice θ se $M \geq C_0 K \log\frac{N}{K}$. Se la matrice di rappresentazione ψ è ortonormale e $M \geq C_0 K \log\frac{N}{K}$, allora la θ rispetta la proprietà *RIP*, se anche ϕ rispetta una delle tre condizioni viste sopra.
Si può definire la ϕ in maniera più deterministica, cercando di renderla più incoerente possibile rispetto la matrice ψ, in modo da necessitare di meno osservazioni per poter ricostruire il segnale di interesse, secondo le formulazioni riportate di seguito:

$$\mu(\phi, \psi) = \sqrt{N} * max_{i,j} <\underline{\phi}_i, \underline{\psi}_j>$$

$$M \geq C * \mu^2(\phi, \psi) * K * \log N$$

Implementazione

In questa sezione si va a descrivere dettagliatamente l'implementazione di diversi algoritmi di *compressed sensing* sviluppati in ambiente *Matlab***,** utili per la ricostruzione del segnale sparso a partire dalle *M* misurazioni.

Iterative Soft Thresholding Algorithm (ISTA)

Uno dei possibili metodi di risoluzione del problema di ricostruzione del segnale sparso *s*, è definito dalla formulazione del problema lasso che viene definito nel seguente modo:

$$\mathrm{argmin}_{\underline{s}} \parallel \theta \underline{s} - \underline{y} \parallel_2^2 + \lambda \parallel \underline{s} \parallel_1$$

La risoluzione avviene tramite l'utilizzo dell'operatore di *proximal gradient* che su problemi con termine di regolarizzazione norma *1* definisce l'operatore di *Soft Thresholding,* il quale permette una selezione sparsa delle caratteristiche mandando a *0* alcuni coefficienti e inducendo quindi sparsità. La soluzione viene riportata di seguito:

$$\underline{x}^{(k+1)} = S_{\lambda t}(\underline{x}^{(k)} - t\nabla g(\underline{x}^{(k)}))$$

$$\underline{s}^{(k+1)} = S_{\lambda t}(\underline{s}^{(k)} - 2t\theta^T(\theta \underline{s}^{(k)} - \underline{y}))$$

Tale operazione viene iterata più volte fino ad arrivare a convergenza (o l'errore di ricostruzione è sufficientemente piccolo e ci si arresta, o si raggiunge il massimo numero di iterazioni definito).
Il codice *Matlab* implementato per realizzare l'algoritmo viene riportato di seguito:

```
% Function used for Iterative Soft Thresholding algorithm

function [s, error_vector, count] = ISTA(y, theta, K)

% theta matrix dimensions
M = size(theta, 1);
N = size(theta, 2);

t = 1 / norm(theta) ^ 2;   % Assign a value to the t step (1 / norm(theta) ^ 2 is a common value)
lambda = 0.01;             % Assign a value to the regularization term

sk_1 = zeros(N,1);    % Initialize the sparse vector to be estimated
error_vector = [];    % Holds the error iteration by iteration
count = 0;            % Keep track of the number of iterations

for i = 1:K
    count = count + 1;
    grad = 2 * theta' * (theta * sk_1 - y);            % Calculate the gradient
    sk = sk_1 - t * grad;                              % Update sk value
    st = (max(abs(sk) - t * lambda, 0)) .* sign(sk);   % Define soft thresholding
    error_vector(i) = norm(theta * st - y);            % Calculate the error

    sk_1 = st;
    % If the error is small enough exit the loop
    if error_vector(i) < 1e-6
         break;
    end
end
s = sk_1;
end
```

L'algoritmo *ISTA* viene implementato dalla funzione *ISTA* che prende in input il segnale *y* da cui ricostruire il vettore sparso, la matrice θ e il livello di sparsità *K*. Si definisce quindi una prima inizializzazione dei parametri utili nella risoluzione del problema. In particolare, si scelgono dei valori di *t* (*step size*) e λ (termine di regolarizzazione) specifici che si è verificato essere ben condizionati in questo caso di studio, con *step size* sufficientemente piccolo da garantire convergenza, ma non troppo grande da non riuscire mai a convergere alla soluzione. L'algoritmo viene implementato iterativamente semplicemente realizzando ciò che si è spiegato in precedenza:

- Si calcola il gradiente della funzione costo;
- Si aggiorna la soluzione al passo precedente rispetto allo *step size* e il gradiente calcolato e si applica l'operatore di *Soft Thresholding* sul risultato;
- Si riporta l'errore calcolato ad ogni iterazione e, se risulta essere sufficientemente piccolo, si assume che l'algoritmo sia arrivato a convergenza;

Si sceglie un numero di iterazione massimo pari a *K* per garantire un buon compromesso tra precisione nella soluzione trovata e velocità di convergenza. Al termine dell'algoritmo si restituisce l'errore registrato nel corso delle varie iterazioni, il vettore *s* stimato e un contatore che tiene traccia del numero di iterazioni necessarie per arrivare a convergenza.

Iterative Hard Thresholding (IHT)

L'algoritmo *IHT* ricerca il vettore sparso sfruttando la funzione di *fixed point*, che, se applicata ad un certo valore lo restituisce invariato, basandosi sulle seguenti relazioni:

$$\theta \underline{s} = \underline{y}$$

$$\theta^T \theta \underline{s} = \theta^T \underline{y}$$

Poiché la quantità $\theta^T \theta$ non è invertibile (la matrice non è a rango pieno, avendo assunto *M* < *N*), si usa un piccolo stratagemma per risolvere il problema:

$$\underline{s} + \theta^T \theta \underline{s} = \underline{s} + \theta^T \underline{y}$$

$$\underline{s} = (I + \theta^T \theta)\underline{s} + \theta^T \underline{y}$$

$$f(\underline{s}) = \underline{s} + \theta^T (\underline{y} - \theta \underline{s})$$

L'obiettivo dell'algoritmo è proprio quello di determinare il valore di *s* che, se passato per la funzione di *fixed point,* rimane invariato. Lo pseudocodice generale dell'algoritmo è il seguente:

Initialize $\underline{s_0} = \underline{0}$

Repeat until $\{ error < \varepsilon \}$

$$\underline{s_n} = H_k(\underline{s_n} + \theta^T (\underline{y} - \theta \underline{s_n}))$$

End repeat

Output $\underline{s}^* = \underline{s_{STOP}}$

Da notare la presenza della funzione H_k che permette di fare *Hard Thresholding*, mantenendo solo le *k* componenti più forti, mentre le altre le azzera garantendo sparsità.

Si realizza il seguente codice *Matlab* che implementa la funzione *IHT*:

```
% Function used for Iterative Hard Thresholding algorithm

function [s, error_vector, count] = IHT(y, theta, K)

% theta matrix dimensions
M = size(theta, 1);
N = size(theta, 2);

s = zeros(N, 1);          % Initialize the sparse vector to be estimated
u = 0.5;                  % Influence coefficient used for updating s
error_vector = [];        % Holds the error iteration by iteration
count = 0;                % Keep track of the number of iterations

for i = 1:K

    count = count + 1;
    s_new = s + u * theta' * (y - theta * s);

    % Sorting in descending order to get the largest values
    [~, index] = sort(abs(s_new), 'descend');

    % Mantain only the largest values
    s_new(index(K+1:end)) = 0;

    error = norm(y - theta * s_new);    % Calculate the error
    error_vector(i) = error;            % Store the error in the vector

    s = s_new;                          % Update the s value

    % If the error is small enough exit the loop
    if error_vector(i) < 1e-6
        break;
    end

end
end
```

La funzione prende in input il segnale *y*, la matrice θ e il livello di sparsità *K* del segnale. Prima vengono inizializzati i valori di interesse, tra cui il coefficiente μ che definisce l'update della soluzione *s* iterazione per iterazione. In questo caso si è scelto di impostarlo a *0.5* garantendo una buona convergenza dell'algoritmo. Il ciclo *for* permette di aggiornare la soluzione *s* fino ad arrivare a convergenza. Il massimo numero di iterazioni che si è deciso di impostare è rappresentato proprio dalla sparsità *K* del segnale, in quanto l'algoritmo *IHT* tende in genere a convergere abbastanza rapidamente e inoltre non si vuole appesantire troppo l'esecuzione dello stesso definendo un numero eccessivo di iterazioni. Ad ogni iterazione si eseguono i seguenti passi:

- Si incrementa il contatore che tiene conto del numero di iterazioni eseguite prima di arrivare a convergenza;
- Si calcola il valore di *s* tale per cui la funzione di *fixed point* descritta precedentemente è soddisfatta;
- Si ordinano gli elementi in maniera decrescente, in modo da mantenere solo i primi *K* valori (quelli di maggior peso) andando a fare *Hard Thresholding* sul vettore sparso stimato;

- Si registra poi l'errore misurato verificando se questo è sufficientemente piccolo da assumere la convergenza dell'algoritmo alla soluzione d'interesse (nel caso in cui la condizione risulta essere verificata si esce dal *loop* anticipatamente, terminando l'algoritmo).

La funzione restituisce poi alla fine il valore *s* stimato, il vettore che tiene conto dell'evoluzione dell'errore attraverso le varie iterazioni e il contatore del numero di iterazioni.

Orthogonal Matching Pursuit (OMP)

L'algoritmo *OMP* cerca la soluzione ottima basandosi sul fatto che le colonne della matrice θ devono essere più ortogonali possibili. Questa considerazione deriva dalla seguente relazione:

$$\delta_k \leq (K-1) * \mu(\theta)$$

Se la condizione sulle colonne della matrice θ è garantita, allora si ottiene un $\delta_k \leq 0$ ($\mu(\theta)$ nullo essendo le colonne incoerenti). Ciò garantisce sempre la validità della proprietà *RIP* e la possibilità di ricostruire il segnale sparso *s* anche in presenza di rumore. La soluzione sparsa in questo caso viene calcolata ricercando le colonne della matrice θ che sono maggiormente correlate con la quantità $e_l = y - \theta s_l$, dove s_l è la soluzione approssimata a *l* componenti della quantità *s*. Queste sono proprio le colonne di θ maggiormente correlate con il residuo definito e sono quelle che spiegano meglio la quantità *y*. Ad ogni iterazione dell'algoritmo si considera l'indice della colonna di θ che ha maggiore energia nella direzione del residuo e_l e si calcola poi il vettore *s* sparso come soluzione di un problema *least square* considerando come supporto (insieme di coefficienti dove il vettore sparso *s* è diverso da *0*) l'insieme degli indici delle colonne di θ fin lì determinato. Si definisce quindi lo pseudocodice dell'algoritmo:

Initialize $S_0 = 0, \underline{s} = \underline{0}$

Repeat until $\{ error < \varepsilon \}$

$$j_{l+1} = \text{argmax}_j\{[|\theta^T \left(\underline{y} - \theta \underline{s}_l\right)|]_j\}$$

$$S_{l+1} = S_l + \{j_{l+1}\}$$

$$\underline{s}_{l+1} = \text{argmin}_z\{\| \underline{y} - \theta \underline{z} \|_2^2, \qquad support(\underline{z}) \in S_{l+1}\}$$

End repeat

Output $\underline{s}^* = \underline{s}_{STOP}$

Il codice *Matlab* realizzato viene riportato di seguito:

```
% Function used for Orthogonal Matching Pursuit algorithm

function [s, error_vector, count] = OMP(y, theta, K)

% theta matrix dimensions
M = size(theta, 1);
N = size(theta, 2);

s = zeros(N,1);       % Initialize the sparse vector to be estimated
s0 = [];              % Mantain the columns of theta useful for reconstruction
el = y;               % Residual value updated every iteration
count = 0;            % Keep track of the number of iterations
error_vector = [];    % Holds the error iteration by iteration
pos = [];             % Mantain the indexes of the columns of the theta matrix

for i = 1:K
    count = count + 1;
    for j = 1:N
        value(j) = abs(theta(:, j)' * el);         % Calculate all the products
    end

    [~, index] = max(value);                       % Retrieve only the maximum value
    s0 = [s0, theta(:,index)];                     % Update the vector s0 with the new column
    theta(:, index) = zeros(M,1);                  % Set this column of theta to 0

    ls = inv(s0' * s0) * s0' * y;                  % Resolve the Least squares problem on the defined suport
    el = y - s0 * ls;                              % Update residual for the next iteration
    error_vector(i) = norm(el);                    % Mantain the error of the iterations
    pos(i) = index;                                % Mantain the index of the column of theta choosed

    if error_vector(i) < 1e-6                      % If the error is small enough exit the loop
         break;
    end
end
s(pos) = ls;                                       % Reconstructed vector
end
```

La funzione *OMP* realizza l'algoritmo *OMP* prendendo in input i valori *y* del segnale di misurazione, la matrice θ e il livello di sparsità definito. Dopo una prima solita inizializzazione di parametri, in cui si inizializza il valore e_l pari al segnale *y* (avendo un vettore *s* iniziale nullo) e si definisce un vettore *pos* che andrà a immagazzinare gli specifici indici di θ scelti dall'algoritmo, si realizza la parte di iterazione e di approssimazione della soluzione con un ciclo *for*. Ad ogni iterazione si svolgono i seguenti passi:

- Si incrementa il contatore per tenere traccia del numero di iterazioni che si sono realizzate;
- Si calcolano tutti i possibili prodotti tra le colonne di θ e il residuo e_l andando poi a scegliere la colonna più importante nel rappresentare *y* (quella relativa al massimo prodotto);
- Si aggiorna la variabile *s0* con la colonna selezionata della matrice θ facendo poi in modo che nella successiva iterazione la colonna scelta di quella stessa matrice non venga più presa in considerazione (la si pone pari al vettore nullo in modo da non rischiare nelle successive iterazioni di riconsiderarla più volte);
- Si va a risolvere il problema *least square* sul supporto caratterizzato, aggiornando passo dopo passo il residuo e_l;
- Viene tenuta traccia dell'errore misurato ad ogni iterazione e se viene verificata la condizione per cui tale valore calcolato è sufficientemente piccolo, si esce dal ciclo.

Anche in questo caso si è ritenuta la scelta di un numero massimo di *K* iterazioni per l'algoritmo soddisfacente per assicurare che la soluzione caratterizzata risulti essere abbia una sparsità esattamente pari a *K*. La funzione sviluppata riporta poi i soliti valori di errore, vettore *s* stimato e contatore per vedere il numero di iterazioni impiegate prima di assumere l'algoritmo terminato.

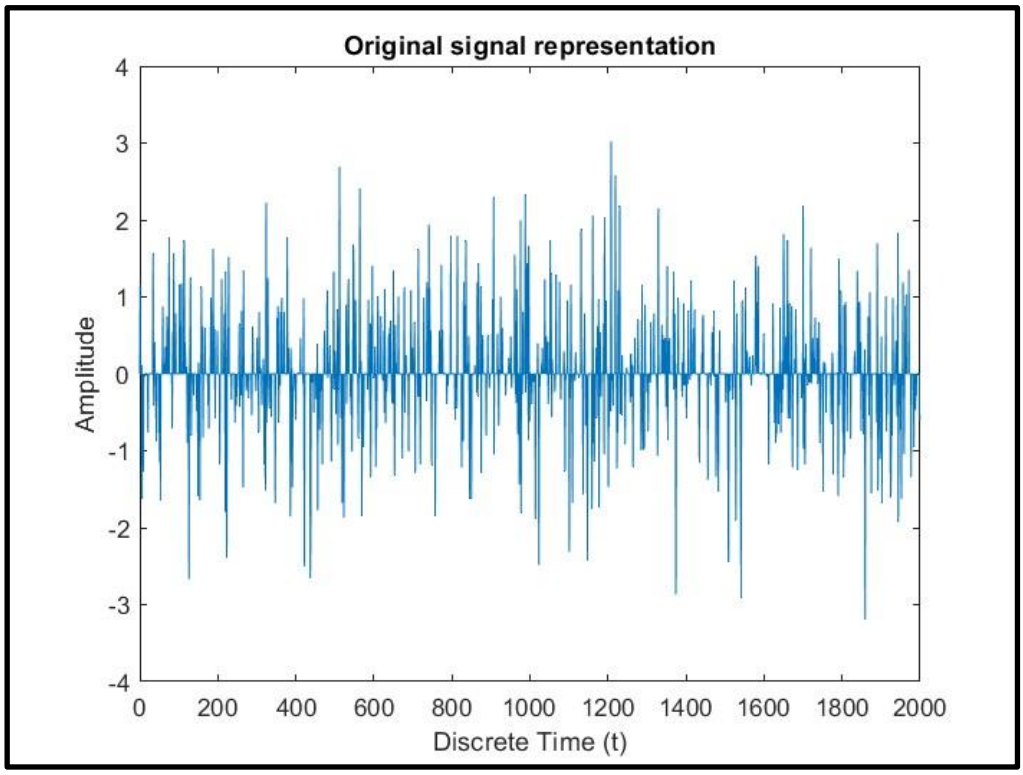
Sperimentazione

In questa sezione si andrà a descrivere il funzionamento di un semplice script *Matlab* che ha l'obiettivo di verificare il corretto funzionamento degli algoritmi elaborati, concentrando l'attenzione sui parametri prestazionali degli stessi. Per prima cosa si inizializzano i valori di interesse:

```
close all
clc
clear
warning off

N = 2000;                      % Dimension of the sparse vector
K = 600;                       % Sparsity level
M = 2 * ceil(K * log(N / K));  % Number of observations

rng(10);  % Set seed for reproducibility

index = randperm(N);  % Define a random permutation of indices from 1 to N
x_orig = zeros(N, 1);

rng(11);

x_orig(index(1:K)) = randn(K, 1);   % Define a k-sparse representation
```

Si definisce quindi la dimensione del segnale *x* di prova, il livello di sparsità *K* (numero di elementi diversi da *0* del vettore) e il numero di osservazioni *M* (la matrice θ rispetta la proprietà *RIP* se vale la condizione per cui $M \geq C_0 K \log \frac{N}{K}$, che assicura quindi la possibilità di ricostruire *s* sparso correttamente). La funzione *rng()* permette di inizializzare il generatore di numeri casuali con un valore noto per garantire la riproducibilità dell'esperimento, assicurando che ad ogni esecuzione dello *script* vengano generati sempre gli stessi valori randomici.

La funzione *randperm(N)* genera una permutazione di *N* elementi, utile poi per caratterizzare il vettore *x* originale come vettore *K-sparso*. Questo mi assicura la presenza di una rappresentazione sparsa per *x*. Viene riportata di seguito la rappresentazione del segnale sparso originale:



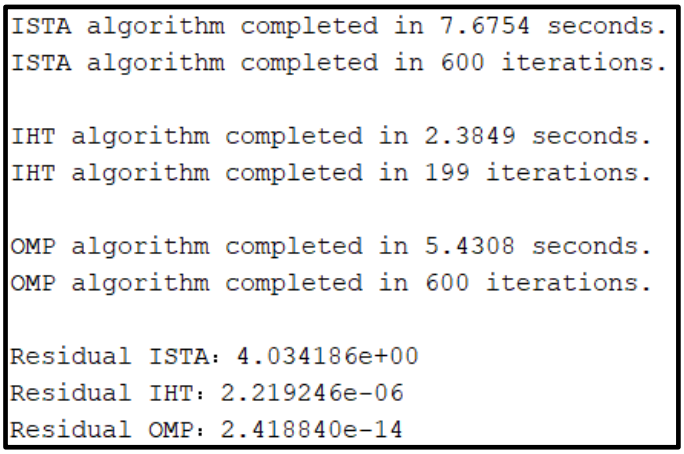
Già a prima vista si vede che sono molti i coefficienti posti a *0* che confermano la definizione di un segnale sparso. Si vanno quindi a caratterizzare tutte le variabili che riguardano la teoria che sta dietro il *compressed sensing*:

```
26    psi = eye(N);           % Define the basis in which the signal is sparse
27    phi = randn(M,N) / sqrt(M);       % Define the measurement matrix with variance 1 / M
28    phi = orth(phi')';      % Ortogonalize the measurement matrix
29    theta = phi * psi;      % Define the theta matrix
30    y = phi * x_orig;       % Simulate measurements
```

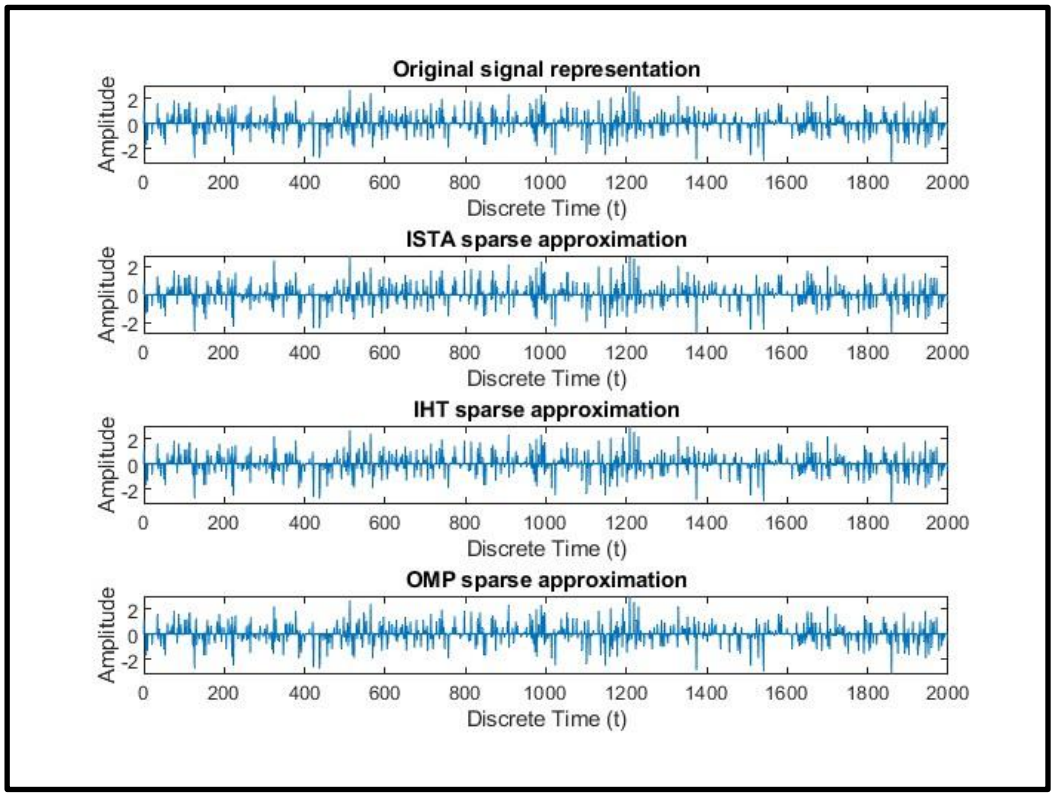
La matrice ψ viene creata come matrice identità essendo il segnale originale già sparso di per sé per come si è inizializzato. Si decide inoltre di descrivere ϕ come la matrice caratterizzata da campioni di una distribuzione gaussiana a varianza $\frac{1}{M}$, che è una delle tre scelte che si fanno comunemente nella realizzazione di algoritmi di *compressed sensing* e che in genere porta a risultati soddisfacenti sperimentalmente. Si sceglie di ortogonalizzare la matrice in questione, dato che nella teoria del *compressed sensing* l'avere a che fare con una scarsa coerenza tra le colonne della matrice risulta essere sempre preferibile. La matrice θ è semplicemente il prodotto tra ψ e ϕ. Infine, si riportano le misure *y* che rappresentano una compressione rispetto al vettore sparso originale secondo la formulazione $\underline{y} = \phi * \underline{x}$.

Ponendoci nelle condizioni di studio riportate nelle figure precedenti, si va ora ad esaminare l'esecuzione dei tre algoritmi da diversi punti di vista. Tenendo conto dei tempi di esecuzione

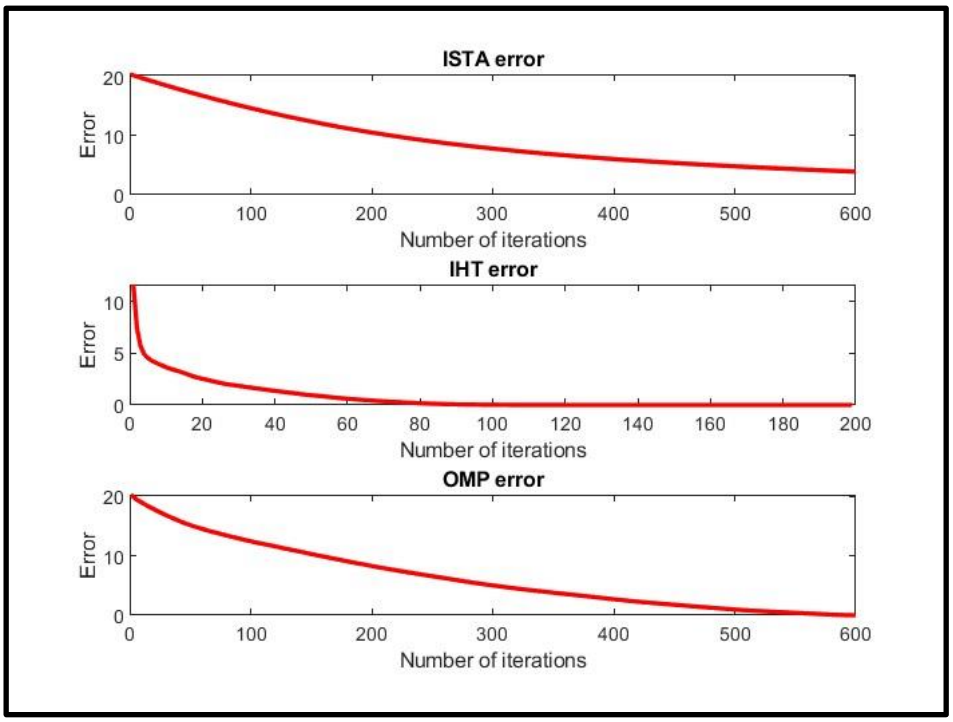
e del numero di iterazioni che occorrono a questi algoritmi per arrivare a convergenza, si denotano i seguenti risultati:



Dai parametri riportati in figura si verifica che tra i tre algoritmi quello che sembra essere più veloce è *IHT* che impiega anche meno iterazioni prima di arrivare a convergere ad un errore sufficientemente piccolo. *OMP* e *ISTA* invece, in termini temporali, sembrano avere lo stesso comportamento, interrompendo l'esecuzione dell'algoritmo solo dopo *K* iterazioni. A partire dal segnale sparso stimato dai tre algoritmi già descritti, si ricostruisce il segnale *x* applicando la formula $\underline{x} = \psi * \underline{s}$. Confrontando i diversi valori di residuo calcolato, che rappresentano la differenza tra il segnale originale e quello ricostruito, si denota in generale che l'algoritmo *OMP* sembra essere quello più accurato, registrando un residuo inferiore rispetto agli altri due casi, mentre l'algoritmo *ISTA* offre una stima di *s* non così precisa, ma comunque accettabile. Il segnale ricostruito nelle varie casistiche viene riportato di seguito e messo a confronto per tutti gli algoritmi impiegati:



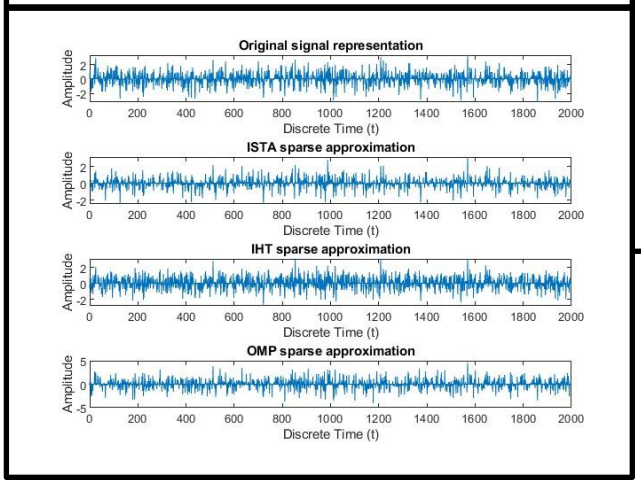
A colpo d'occhio si vede che la ricostruzione è soddisfacente in tutti i casi analizzati: nonostante ISTA abbia il residuo calcolato maggiore rispetto agli altri due casi, sembra comunque definire un segnale *x* abbastanza fedele all'originale. A questo punto si va a rappresentare anche l'andamento dell'errore che viene commesso dagli algoritmi iterazione per iterazione osservandone la convergenza verso valor nullo:

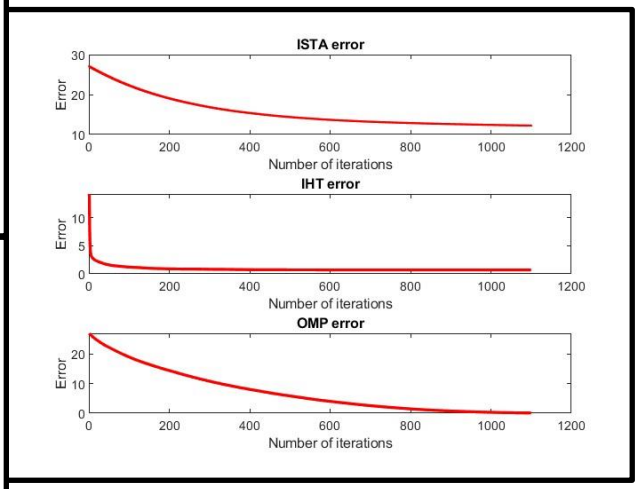


Il fatto che i tre algoritmi in questo caso riescono a convergere molto bene alla soluzione ottima è visibile anche nell'andamento dell'errore. *IHT,* in particolare, è quello che tende più velocemente al valore nullo e dopo poco meno di *200* iterazioni si può considerare concluso, assumendo sia arrivato a convergenza. Gli altri due algoritmi hanno bisogno di più iterazioni per convergere, ma alla fine riescono a garantire un errore abbastanza basso (*ISTA* non riesce a convergere a valori di errore tanto bassi quanto gli altri due algoritmi, ma comunque accettabili). Si è considerato un caso di sperimentazione ben definito, con un certo valore di *K* fissato che determina il numero di coefficienti non nulli del segnale e che va a rappresentare un segnale molto sparso (con molti coefficienti a *0*). Si potrebbe verificare cosa accade all'aumentare di *K* nei tre algoritmi considerati. Si considerano due casi ulteriori di cui si andranno a riportare le stesse osservazioni di interesse fatte in precedenza: *K = 1100 e K = 1700*.

Per K = 1100 si ottengono i seguenti risultati:

```
ISTA algorithm completed in 12.2893 seconds.
ISTA algorithm completed in 1100 iterations.

IHT algorithm completed in 11.9663 seconds.
IHT algorithm completed in 1100 iterations.

OMP algorithm completed in 25.8729 seconds.
OMP algorithm completed in 1100 iterations.

Residual ISTA: 1.905965e+01
Residual IHT: 1.892310e+01
Residual OMP: 2.642063e+01
```

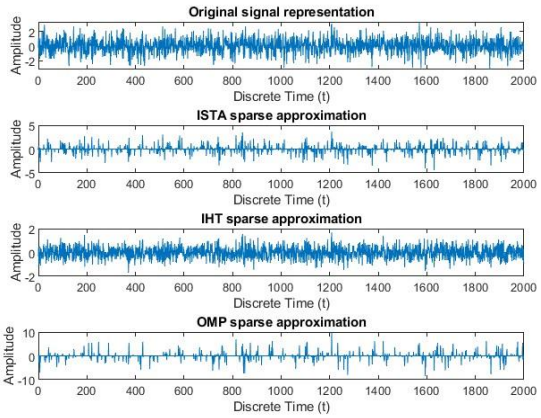


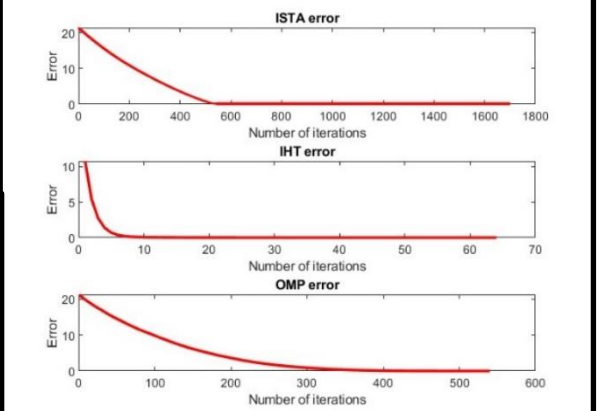


Gli algoritmi *IHT* e *OMP* riescono a convergere alla soluzione con un errore approssimativamente nullo al contrario di *ISTA*. I segnali *x* ricostruiti però in tutti e tre i casi risultano essere molto diversi dall'originale.

Per *K = 1700* valgono le seguenti rappresentazioni:

```
ISTA algorithm completed in 7.0247 seconds.
ISTA algorithm completed in 1700 iterations.

IHT algorithm completed in 0.2783 seconds.
IHT algorithm completed in 64 iterations.

OMP algorithm completed in 2.5247 seconds.
OMP algorithm completed in 540 iterations.

Residual ISTA: 4.014493e+01
Residual IHT: 3.514564e+01
Residual OMP: 5.899799e+01
```





Anche qui valgono le considerazioni fatte in precedenza e vengono rinforzate da un residuo ancora più importante.

Quindi, nei due casi, si assiste ad un considerevole decremento delle prestazioni di tutti gli algoritmi impiegati. Si stanno infatti considerando un maggior numero di coefficienti diversi da *0* che sembrano quindi dare problemi nella ricostruzione del segnale originale. In particolare, l'algoritmo *OMP* sembra essere quello che subisce la maggiore penalizzazione in termini di ricostruzione registrando un incremento di residuo superiore rispetto gli altri due, denotando un segnale *x* stimato molto diverso dall'originale.

Conclusioni

L'obiettivo di questo progetto era quello di verificare l'esecuzione di diversi algoritmi di *compressed sensing* per la ricostruzione di un segnale rappresentabile in maniera sparsa. Durante la fase di sperimentazione si è arrivati ad alcuni risultati interessanti:

- In generale l'algoritmo *OMP* sembra essere quello che lavora meglio per segnali ben condizionati dal punto di vista della sparsità, offrendo un buon tempo di convergenza a fronte di grande precisione nella ricostruzione del segnale originale;
- Gli algoritmi *ISTA* e *IHT*, definiti sugli stessi livelli di sparsità, offrono prestazioni comunque buone riuscendo a convergere, in generale, in tempi minori alla soluzione rispetto *OMP*, ma fornendo una soluzione meno accurata;
- All'aumentare del parametro *K* le prestazioni degli algoritmi tendono a peggiorare rapidamente, in particolar modo per *OMP* che registra il decremento di precisione più importante: quando si hanno dei segnali non ben condizionati dal punto di vista della sparsità, tutti e tre gli algoritmi tendono a lavorare peggio, ma in particolare *OMP* sembra essere quello più sensibile a variazioni del parametro *K* e alla presenza di segnali poco sparsi;
- Nel caso in cui si ha a che fare con segnali che hanno rappresentazioni poco sparse, gli algoritmi di *compressed sensing* non riescono a lavorare bene come quando si ha a che fare con segnali con pochi coefficienti non nulli;
- Mantenendo un livello di sparsità adeguata del segnale comunque tutti gli algoritmi lavorano in modo soddisfacente;
- *ISTA,* tra gli algoritmi analizzati, sembra essere il peggiore in quanto non riesce a convergere ad un valore tanto basso di errore quanto quello degli altri due algoritmi, denotando rappresentazioni più imprecise.

Ovviamente i risultati ottenuti sono influenzati da tanti fattori e tante scelte che si compiono nell'effettuare questa sperimentazione come ad esempio quelle riguardanti la scelta dei parametri λ e *t* dell'algoritmo *ISTA*, o del numero massimo di iterazioni per ogni algoritmo (si sceglie *K* in genere per avere un buon compromesso tra velocità di convergenza e precisione della stima del vettore sparso *s*), o della particolare matrice ψ realizzata (si è considerata in questo caso la distribuzione gaussiana con varianza pari a $\frac{1}{M}$ poi ortogonalizzata, ma si sarebbe potuto utilizzare tutt'altra caratterizzazione). L'analisi effettuata in questo progetto comunque mette in risalto l'importanza di questi algoritmi di *compressed sensing* che garantiscono la ricostruzione di un segnale sparso a partire da un numero inferiore di misurazioni, che in un ambiente vasto e complesso come quello dei *big data*, permette una gestione più efficiente dei dati e delle comunicazioni. La scelta dell'algoritmo dipende molto da quelle che sono le esigenze specifiche dell'applicazione in esame. Ad oggi il *compressed sensing* è molto usato in molti campi e applicato in tante diverse situazioni come nell'*imaging* medico, in cui si cerca di ridurre il numero di misurazioni garantendo comunque una buona qualità dell'immagine, nell'elaborazione del segnale audio, che si evince nella riduzione di dati audio, ad esempio per

la trasmissione di dati in *streaming,* e nelle comunicazioni *wireless*, per il recupero di segnali sparsi, permettendo una trasmissione più efficiente e meno pesante delle informazioni.