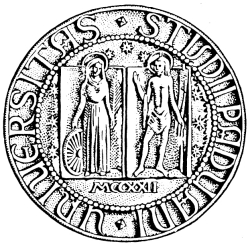


Università degli Studi di Padova
Facoltà di Ingegneria

Attività Formativa: tesi

Implementazione di Algoritmi di Compressione del Segnale per Applicazioni di Monitoraggio Intelligente in Reti di Sensori Radio

Laureando: **Del Re Riccardo**
Relatore: **prof. Rossi Michele**
Corso di Laurea Magistrale in Ingegneria delle Telecomunicazioni

24/04/2012

2011/2012

Indice

Elenco delle figure

Sommario

In questa tesi viene presentata e discussa una nuova tecnica di compressione di segnali per Wireless Sensor Network, basata su Empirical Mode Decomposition (EMD). Inoltre, vengono testati due schemi di compressione noti in letteratura (AR e PLA-LTC) e svolti dei test di performance comparison considerando sia segnali sintetici che tracce di dati reali. EMD è una tecnica di signal processing che decompone segnali rumorosi e non necessariamente stazionari in un certo numero di modi che presentano alta auto-correlazione, i quali possono essere facilmente codificati grazie a semplici modelli di ordine basso. I risultati indicano che EMD, combinato con le tecniche di filtraggio di basso ordine, porta a buone prestazioni in termini di compression ratio e energy consumption.

Capitolo 1

Introduzione

Negli ultimi anni si è notato un sempre maggiore interesse nel campo delle modalità di produzione e consumo di energia elettrica, ed è iniziata la corsa alle energie rinnovabili. Questa soprattutto a causa delle previsioni sul possibile esaurimento delle risorse non rinnovabili, che rappresentano oggi la fonte di oltre l'80% dell'energia prodotta a livello mondiale. I grandi impianti di produzione e distribuzione hanno dovuto infatti adeguarsi inoltre alla costante crescita della domanda energetica. In un futuro ormai prossimo quindi, l'utilizzo delle energie rinnovabili sarà determinante, e di conseguenza, copriranno un ruolo assai importante anche le tecnologie e le strutture adibite alla loro gestione e distribuzione, al fine di garantire la maggior efficienza possibile.

In questa necessaria evoluzione svolge un ruolo centrale la rete elettrica stessa, che è destinata a cambiare, trasformandosi da sistema centralizzato (centrale elettrica $\Rightarrow$ consumatore) a sistema distribuito. L'utente, quindi, sarà in grado di produrre a sufficienza sia energia per il proprio fabbisogno, che un *surplus* energetico, che potrà essere venduto alle società distributrici (ciò che avviene attualmente), o addirittura sfruttato dagli utenti geograficamente vicini.

La rete elettrica tradizionale è nota in letteratura con il nome di "Grid", ovvero una griglia (rete, reticolato), i cui

nodi rappresentano gli utenti (edifici) che consumano energia o centrali che la producono. Queste Grid possono essere suddivise in reti più piccole a media e bassa tensione ("Microgrid"), in grado di scambiare energia elettrica tra di loro o con i grandi impianti di produzione. Si tratta quindi di una struttura a livelli gerarchici, che può essere paragonata ad Internet, in cui i flussi viaggiano in modo bi-dimensionale tra le Microgrid e la "Backbone" della rete principale ad Alta Tensione. Lo scopo è dunque quello di rendere efficiente tale sistema, monitorandone i flussi ai vari livelli, regolandoli di conseguenza, per ridurre il più possibile gli sprechi ed i costi sostenuti dall'utilizzatore finale. Si tratta quindi di rendere intelligente la rete, da cui il termine "Smart grid".

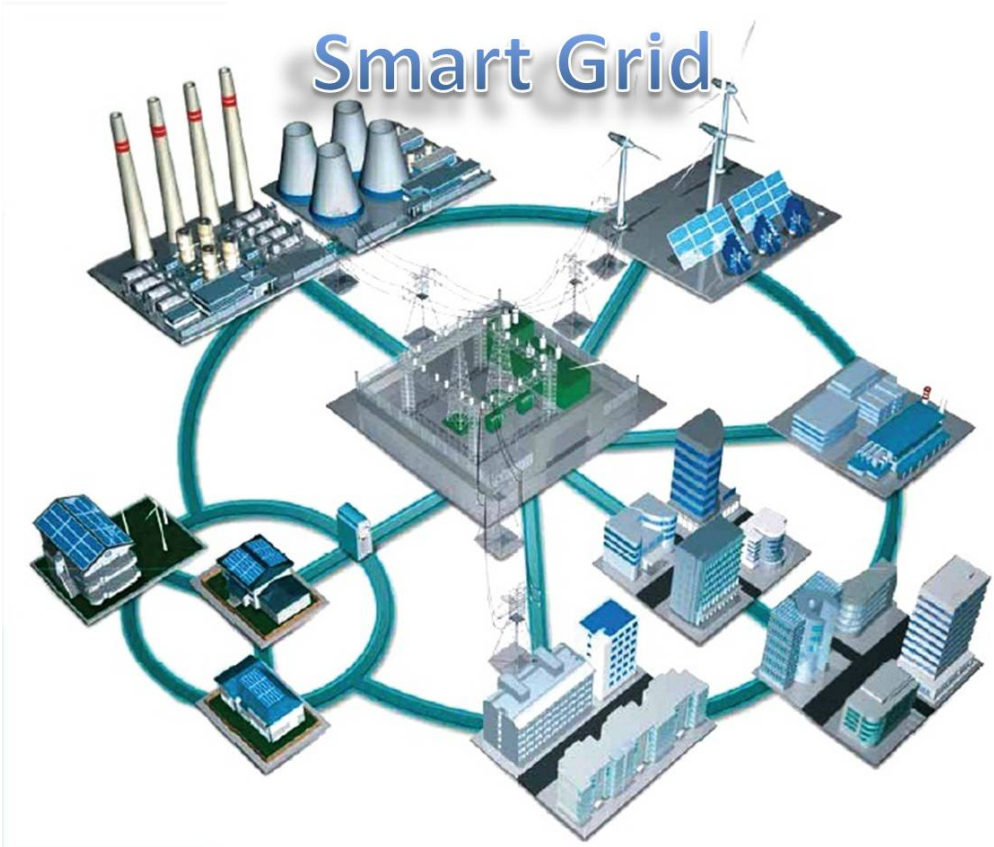


Figura 1.1: Esempio di Smart Grid

Tale scopo può essere raggiunto con l'utilizzo di sensori e controllori ai vari livelli gerarchici. Negli ultimi anni i sensori wireless e le tecnologie mobili hanno visto un grande aumento e diffusione. Il progresso tecnologico nella fabbricazione dei micro-controllori, il design dell'hardware e la loro integrazione in apparati sempre più piccoli e meno costosi hanno reso possibile l'incorporazione delle tecnologie wireless in ogni oggetto. Le tecnologie wireless, inoltre, non richiedono la posa di cavi per la comunicazione, e questo aumenta notevolmente la facilità di installazione in qualsiasi luogo, nonché evita la necessità di lavori pesanti come la rottura di muri in strutture che non prevedevano l'installazione di tali apparecchi. Questo porterà alla creazione di una rete di oggetti senza precedenti, dove ogni oggetto è interconnesso all'altro e insieme connesso ad Internet, dove scambierà informazioni su se stesso e potrà addirittura interagire con l'ambiente circostante. Ciò comporterà un elevato scambio di dati ottenuti dall'acquisizione di parametri ambientali e per la gestione della rete stessa. Le reti di sensori wireless hanno ottenuto un buon livello di maturità e sono tra i componenti più importanti per lo sviluppo dell'Internet of Things: esempi di applicazioni delle WSN sono monitoraggio ambientale, geologico, strutturale e naturalmente smart grid e gestione dell'energia a livello casalingo.

Tutte queste applicazioni richiedono un enorme mole di dati da acquisire ed analizzare. Inoltre, la maggior parte dei sensori vengono dislocati in grandi aree, e questo porta ad una rete ad alta densità di nodi. I dati acquisiti devono poi essere trasmessi via radio, per mezzo di appropriati protocolli di routing, al nodo centrale per essere eventualmente elaborati da dispositivi più performanti. Un primo problema che si presenta in questo scenario è appunto il grande numero di dispositivi necessari a causa del corto raggio di

trasmissione di ciascuno. Le previsioni danno un notevole incremento del numero di questi, e se così fosse, l'ammontare dei dati da gestire dalla rete diventerà proibitivo. Un secondo problema è rappresentato dalle ridotte capacità in termini di memoria, processing, etc. dei dispositivi per IoT e dal fatto che la comunicazione wireless è la causa maggiore del consumo di energia. Solitamente, i nodi sono alimentati a batteria e devono essere progettati per rimanere incustoditi, ma allo stesso tempo operativi per un lungo periodo di tempo. Recentemente, sono state proposte numerose strategie per alleviare il consumo di energia e la dipendenza dalla batteria nelle WSN, ma comunque, limitare il più possibile il consumo di energia rimane un obiettivo importante.

Come mezzo per alleviare questi problemi, negli ultimi anni numerosi ricercatori hanno presentato svariate tecniche. Spesso, i dati acquisiti dai dispositivi vicini sono molto correlati e possono essere efficientemente compressi. La maggior parte dei sistemi di compressione dati per le WSN sono importati da altre tipologie di applicazione, come l'elaborazione audio (MP3), immagini (JPG) o video (MPEG). Inoltre, la maggioranza delle compressioni con perdite sono basate sulla scomposizione del segnale. L'idea è quella di separare il segnale in modo da concentrare le informazioni più importanti in poche componenti.

In questa tesi viene affrontato lo sviluppo di un algoritmo di compressione con perdite, mirando a quelle applicazioni che necessitano di *signal processing* centralizzato, ovvero per i quali non è possibile la completa elaborazione dei dati a livello locale e distribuito. Il fine è quello di una compressione temporale dei segnali acquisiti, in modo da massimizzare l'efficienza energetica e prolungare il tempo di vita del nodo sensore. Viene utilizzata una recente tecnica di *signal processing*, introdotta da Huang et al. [1], detta Empirical

Mode Decomposition (EMD). EMD è una tecnica capace di scomporre segnali affetti da rumore e non necessariamente stazionari in un certo numero di modi (IMFs). Questi modi sono, in genere, caratterizzati da una più alta correlazione rispetto al segnale originale, e per questo possono essere efficientemente codificati da modelli di basso ordine. EMD estrae le oscillazioni del segnale fisico senza fare assunzioni sulla sua statistica. Il procedimento di compressione consiste nell'applicazione di EMD, la selezione di un sottoinsieme di modi sufficienti a ricostruire il segnale originale con un massimo errore di ricostruzione, e l'applicazione di un filtro di basso ordine e computazionalmente efficiente ad ognuno dei modi selezionati (ottenendo una rappresentazione compressa per ognuno di essi). La compressione è quindi raggiunta inviando i parametri dei filtri appena calcolati.[3][9]

Capitolo 2

Compressione tramite EMD

La maggior parte delle applicazioni di rilevamento di dati, quali essi geofisici, ambientali o relative all'attività umana trattano segnali non lineari e processi non stazionari mostrando scale temporali o spaziali intermittenti. I metodi classici per l'analisi dei dati sono basati sulla decomposizione del segnale su una basse fissa, decisa a priori. Dunque, trovare una base adatta richiede una conoscenza a priori del segnale, in particolare della sua scala temporale, e questa informazione è in genere difficile da ottenere nel corso dell'acquisizione. Storicamente, si sono caratterizzati tramite un analisi statistica numerosi tipi di dati, ma i risultati sono applicabili solamente a determinati tipi di domini. Esempi di questi sono il JPEG per la compressione delle immagini e MP3 per la compressione audio.

Empirical Mode Decomposition (EMD) è stata introdotta da Huang et al. [1] per l'analisi di dati generati da processi non stazionari e non lineari. Il vantaggio del suo impiego è che EMD scompone automaticamente il segnale in modi determinati intrinsicamente, detti Intrinsic Mode Function (IMF), che costituiscono una base per il segnale determinata empiricamente. Dunque, la rappresentazione del segnale è totalmente adattiva e la base appropriata al segnale viene trovata in *runtime*.

Gli algoritmi classici basati sulla decomposizione del segnale su base fissa hanno il vantaggio che la base stessa è conosciuta sia lato sorgente che lato destinazione. I compressori con perdite sfruttano questa conoscenza per codificare solo un insieme rappresentativo dei coefficienti della base. L'efficacia di questo metodo dipende dalla compattazione di un insieme di funzioni in un vettore di coefficienti, così da catturare maggior informazione del segnale possibile.

Dato che in EMD la base è estratta dal segnale stesso, può essere conosciuta solo lato sorgente. Questo richiederebbe la trasmissione dell'intera base invece di un piccolo insieme di coefficienti, portando quindi non solo ad una non compressione, ma addirittura ad un incremento del numero di dati da trasmettere. Tuttavia, un analisi più dettagliata delle caratteristiche dei modi che compongono il segnale rivelano un modo più intelligente di utilizzare EMD. Infatti, i modi sono altamente auto-correlati e quindi possono essere efficientemente compressi grazie a modelli a bassa complessità e successivamente si ottiene la compressione grazie all'invio dei coefficienti di tali modelli.

Il maggior pregio di EMD è la sua adattività, ovvero, non fa assunzioni sulla frequenza in gioco e funziona indipendentemente da questa. Per costruzione, gli IMFs sono segnali oscillanti continui, con transizioni dolci dai massimi ai minimi e viceversa, inoltre presentano media nulla. Si comportano come un segnale pseudo-sinusoidale con modulazione sia in frequenza che in ampiezza. Questa particolare caratteristica si riflette sulla loro alta auto-correlazione.

2.1 Introduzione

L'algoritmo di compressione adottato è, come già citato, una tecnica recente, conosciuta come *Empirical Mode Decompo-*

sition, proposta nel 1998 da N.E.Huang et al.[1] al fine di rappresentare segnali non stazionari come somma di componenti AM-FM a media nulla. Come specificato nel nome, sebbene questo metodo dia spesso risultati corretti, la tecnica va incontro alla difficoltà di essere definita da un algoritmo, e quindi non ammette formulazioni analitiche che permetterebbero un analisi teoretica e una valutazione delle performance. L'algoritmo vero e proprio verrà analizzato successivamente.

I risultati dell'EMD, come detto, sono dei segnali a media nulla, detti IMF (Intrinsic Mode Function), ed ognuno di essi, oltre ad avere le caratteristiche sopra citate, presenta una correlazione più o meno forte tra i suoi campioni. Per questo si andranno quindi a scegliere gli IMF con più correlazione, scartando quelli meno correlati che, a fronte di risultati teorici nonché sperimentali, saranno le componenti di rumore del segnale originale. Questo implica una sorta di prefiltraggio del segnale. Successivamente si addotterà a scelta un modello di predizione AR (a 2 o 3 coefficienti) o uno di predizione lineare, sui singoli IMF così da ridurre notevolmente la mole di dati da inviare sul canale wireless. Come detto all'inizio di questo paragrafo, non vi è una regola per cui l'algoritmo in realtà funziona. Tramite simulazione Matlab si è infatti visto che si arriva a compressioni anche del 20% del carico iniziale, a patto che il segnale soddisfi determinate caratteristiche sulla correlazione e sulla lunghezza. Sempre sperimentalmente si è osservato che più è grande l'ampiezza del rumore, meno l'algoritmo comprime. Si può quindi delineare in quali applicazioni sia possibile utilizzare tale algoritmo. Si andrà nel dettaglio di queste considerazioni più avanti.

In Figura 2.1 è presentato lo schema del progetto di tesi, il quale si compone di cinque blocchi principali. Il primo blocco è costituito dall'algoritmo EMD, il quale scompone il segnale in componenti AM-FM a media nulla detti IMFs (Intrinsic

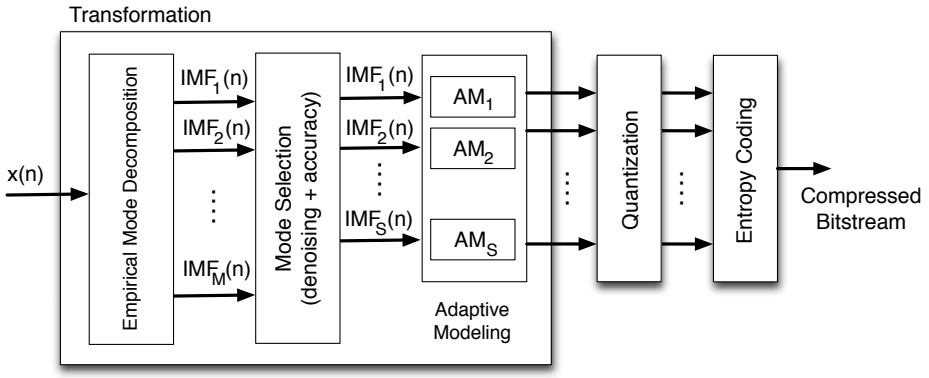


Figura 2.1: Schema a blocchi del progetto di tesi

Mode Functions). Questo algoritmo verrà descritto nel dettaglio nella sezione 2.2. Il secondo blocco è la selezione dei modi: ogni IMF ritornato dall'algoritmo EMD presenta una lunghezza di correlazione tra i campioni, la quale viene usata come criterio di selezione, in quanto i modi con lunghezza di correlazione più corta sono considerati rumore, e di conseguenza vengono, da qui in poi, trascurati; verranno quindi considerati solo S modi sugli M iniziali, effettuando dunque un filtraggio del segnale originale e abbattendo il carico di dati da elaborare successivamente. Questa parte verrà curata nella sezione 2.3. Il blocco successivo si basa sulla creazione di un modello adattivo (Adaptive Modeling, AM), in grado di approssimare il segnale in esame (i vari IMFs), diminuendo notevolmente la quantità di informazione per la ricostruzione del segnale. In questo progetto si sono considerate due tecniche, un modello di predizione di tipo AutoRegressive (AR) e un modello di predizione di tipo lineare Piecewise Linear Approximation - Lightweigth Temporal Compression (PLA-LTC), molto note in letteratura [10] [7]. L'analisi dettagliata di tali modelli verrà presentata nella sezione 2.4.1 per quanto riguarda il modello AR, e nella sezione 2.4.2 per LTC. Il quarto e il quinto blocco rappresentano, rispettiva-

mente, la quantizzazione dei dati ricavati al punto precedente e la codifica entropica. In questo progetto non sono state curate queste sezioni, in quanto esse sono già implementate nel sensore utilizzato per l'acquisizione, elaborazione e infine trasmissione dei dati.

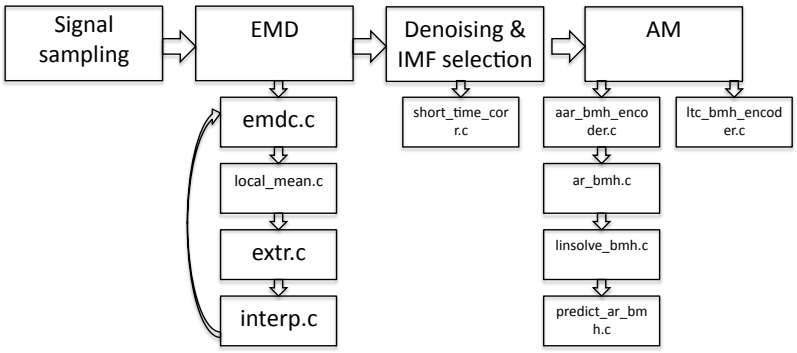


Figura 2.2: Schema delle routine C

In Figura 2.2 è riportato lo schema sequenziale delle routine C (in appendice) per il calcolo della compressione, riferite allo schema a blocchi di Figura 2.1. Le routine svolgono le seguenti funzioni (per il dettaglio delle funzioni vedere nella parte seguente del capitolo):

- *emdc.c*: la "classe" generale dell'algoritmo EMD; inizializza i vettori utili per il calcolo dei modi, chiama le funzioni per il calcolo della media, controlla le condizioni di stop e aggiorna i risultati delle operazioni interne.

- *local_mean.c*: tramite la funzione *mean_and_amplitude* chiama *extr.c* per il calcolo degli estremi e le boundary conditions e infine chiama *interpolation.c* per l'interpolazione dei massimi e dei minimi.

- *extr.c*: per mezzo della funzione *extr* calcola i massimi e i minimi del vettore passato come argomento, men-

tre grazie alla funzione *boundary_conditions* risolve il problema delle condizioni al contorno.

- *interp.c*: interpola i dati passati come argomento secondo interpolazione spline cubica.
- *short_time_corr.c*: calcola la lunghezza di correlazione della sequenza passata come argomento.
- *aar_encoder.c*: è la "classe" generale per il calcolo del modello AutoRegressivo; inizializza i vettori da passare come argomento alle funzioni successive e confronta il segnale originale con il segnale ricostruito.
- *ar.c*: è la funzione "core" del calcolo del modello AR; vengono inizializzate le matrici e i vettori che verrano poi passati alla funzione *linsolve* per il calcolo dei coefficienti del filtro AR.
- *linsolve.c*: calcola, mediante inversione di una matrice, i coefficienti del filtro predittivo.
- *predict_ar.c*: ricostruisce il segnale grazie ai coefficienti del filtro calcolati in *ar*; il segnale ricostruito servirà poi per il controllo sull'errore eseguito nella funzione *aar_encoder*.
- *ltc_encoder.c*: è l'encoder per l'algoritmo LTC; semplicemente crea i segmenti interpolanti i punti in esame e restituisce il modello.

In Figura 2.3 vengono rappresentati il segnale originale e il segnale ricostruito dopo l'esecuzione dell'algoritmo EMD (ottenuto tramite simulazione Matlab), considerando quindi tutti i modi, anche quelli ritenuti rumore. In ascissa si ha il campione n-esimo, mentre in ordinata il valore di ampiezza del segnale nell'istante n-esimo. Come si osserva si ha una

perfetta ricostruzione del segnale originale, e questo conferma la bontà dell'algoritmo EMD per la scomposizione dei segnali.

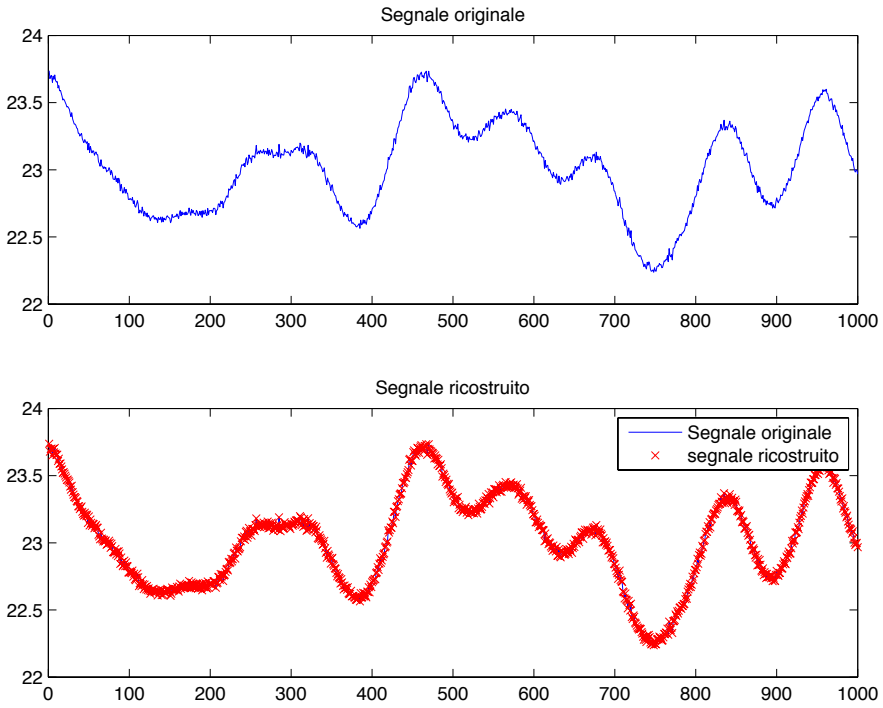


Figura 2.3: Segnale originale e segnale ricostruito considerando tutti i modi

2.2 Algoritmo EMD

Il punto di partenza dell'EMD è di considerare le oscillazioni del segnale a livello locale. È possibile cogliere le oscillazioni a frequenze diverse che compongono il segnale anche solamente osservandone l'andamento. Ad esempio una componente sommata ad una a frequenza più bassa a sua volta sommata ad un'altra e così via. Il metodo migliore è quello di osservare tra due massimi o minimi locali, che in via sistematica viene chiamato *sifting process*. L'algoritmo effettivo dell'EMD viene riassunto in:

1. Trovare gli estremi locali di $x(t)$, dove $x(t)$ è il segnale in ingresso all'algoritmo, dunque il segnale da scomporre.
2. interpolare, usando spline cubica, i minimi (massimi) trovando quindi il lower (upper) envelope $e_{min}(t)$ ($e_{max}(t)$).
3. Calcolare la media $m(t) = \frac{e_{max}(t)+e_{min}(t)}{2}$.
4. Trovare il dettaglio $d(t) = x(t) - m(t)$.
5. Ripetere il processo su $d(t)$ fino ad una condizione di stop.
6. Trovato il primo modo esso è uguale a $d(t)$.
7. Calcolare il residuo $r(t) = r(t) - d(t)$ ($r(t) = x(t)$ nell'inizializzazione).
8. Ripetere i passi fino alla condizione di stop finale (aggiornando $x(t)$ a $r(t)$).

Per costruzione, il numero di estremi decresce ad ogni iterazione ed è garantito che la decomposizione è completa in un numero finito di modi. La selezione dei modi corrisponde ad un filtraggio automatico e adattivo tempo-invariante.

La Figura 2.4 mostra il risultato di una tipica scomposizione EMD di una serie $x(n)$. La prima colonna (a) rappresenta i sette modi in cui $x(n)$ viene scomposta; in ascissa è riportato l'istante temporale n-esimo, mentre in ordinata il valore in ampiezza del singolo modo. La seconda colonna (b) mostra il segnale originale $x(n)$ (linea continua) e il segnale ricostruito $\hat{x}(n)$ (linea tratteggiata), dove $n = 1, 2, \ldots, 500$ è l'indice temporale; in particolare, il grafico nella i-esima riga della colonna (b) mostra la differenza tra $x(n)$ e $\hat{x}(n)$ quando sono usati i primi i modi per la ricostruzione, con $1 \leq i \leq 7$. La terza colonna (c) rappresenta l'autocorrelazione del modo corrispondente in colonna (a). Si può osservare come i

primi quattro modi siano sufficienti per rappresentare accuratamente la sequenza in ingresso $x(n)$. Inoltre, questi quattro modi sono molto correlati, mentre i tre successivi possono essere scartati in quanto contengono oscillazioni che assomigliano a rumore. In Tabella 2.1 viene mostrata l'autocorrelazione per ogni IMF e il Root Mean Square Error (RMSE) associato alla ricostruzione del segnale quando è utilizzato l'i-esimo modo, con $i = 1, \ldots, 7$.

Come anticipato, l'algoritmo EMD dipende da un numero diverso di opzioni che devono essere decise e controllate dall'utente a priori e richiedono competenza e conoscenza delle possibili conseguenze, al fine di ottenere il miglior risultato possibile a fronte di una complessità di calcolo minore.

L'operazione base nel EMD è il calcolo dell *upper envelop* e del *lower envelop*. La natura dell'interpolazione scelta gioca un ruolo importante. Come raccomandato da Huang et al. [1] e confermato da Rilling et al. [2], il metodo di interpolazione migliore risulta la spline cubica. Altri tipi di interpolazione (lineare o polinomiale) tendono ad accrescere il numero di iterazioni di *sifting* e a "sovradecomporre" i segnali, separando le componenti su modi adiacenti. In Figura 2.5 viene presentato un estratto grafico dell'operazione di interpolazione. Sono facilmente distinguibili il *lower* e l'*upper envelope* (in rosso), ottenuti tramite interpolazione

Modi	RMSE	CorrLen
1	1.039	170
2	0.814	92
3	0.389	34
4	0.110	19
5	0.105	5
6	0.090	3
7	0.000	2

Tabella 2.1: RMSE e lunghezza di correlazione per il segnale di Figura 2.4

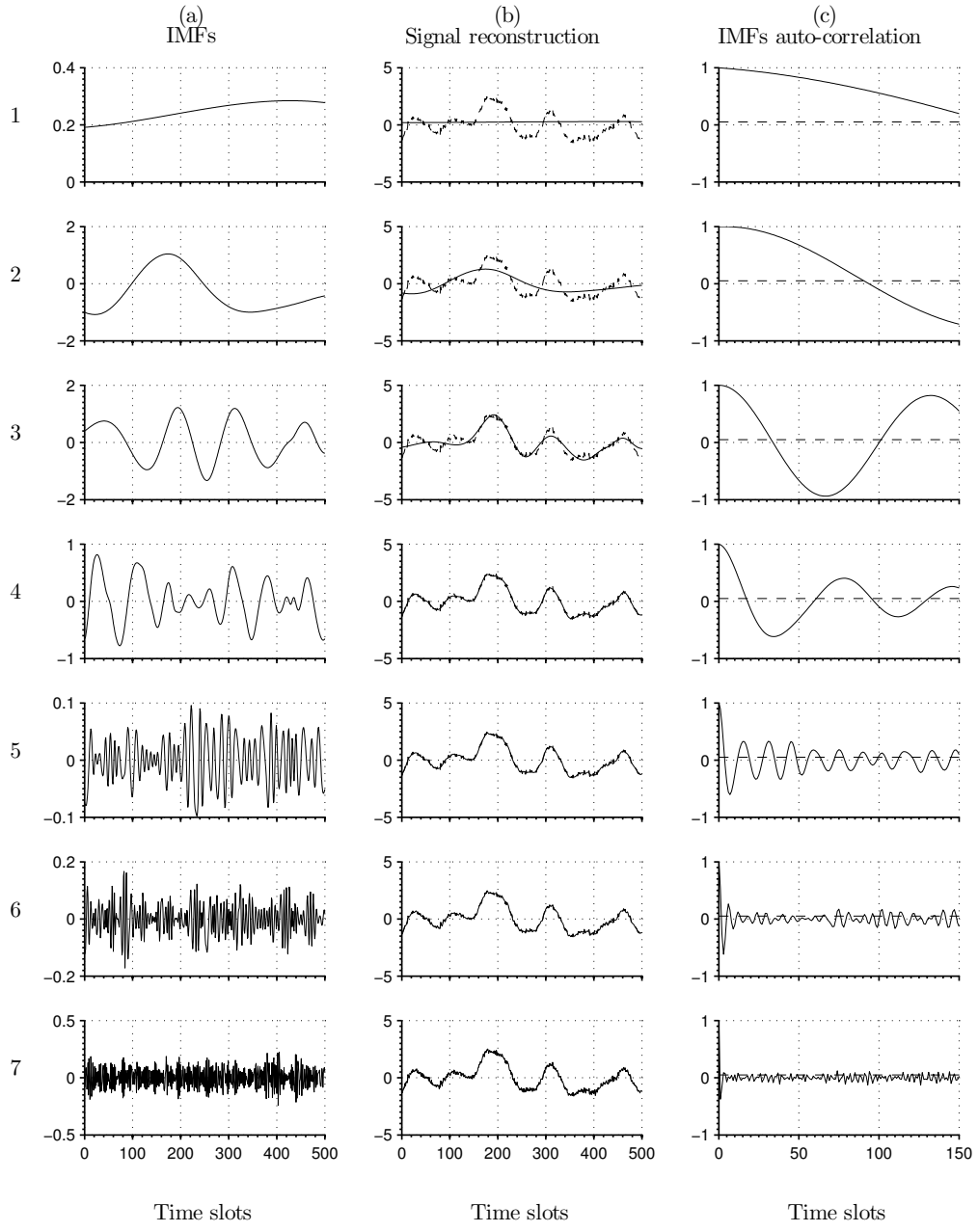


Figura 2.4: Scomposizione del segnale in modi

spline cubica tra i massimi (*upper*) e i minimi (*lower*) locali del segnale in esame; inoltre è raffigurata la media tra i due inviluppi (in azzurro), che è quella che successivamente andrà

sottratta al segnale in esame (in blu).

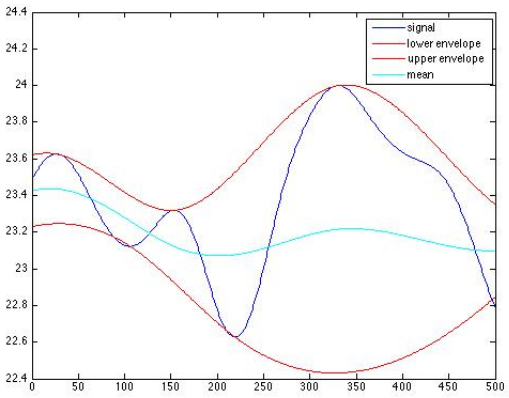


Figura 2.5: Esempio di interpolazione spline cubica sui massimi e minimi

Un altro aspetto è quello che riguarda le *boundary conditions*. Per costruzione l'algoritmo spline potrebbe creare problemi quando usato in condizioni di una finestra di osservazione finita. Per ovviare a tale problema la scelta migliore è stata quella di "specchiare" i valori di *bordo* al di fuori della finestra di osservazione del segnale, cosicché la spline non risenta delle condizioni al contorno.

Ad esempio, se il vettore dei minimi è composto da

$$e_{min} = [1\ 5\ 10\ 21\ 37\ 48]$$

su una finestra di osservazione di 50 campioni, dopo il "mirroring" risulterà

$$e_{min} = [\underbrace{-10\ \ -5}_{NBSYM}\ 1\ 5\ 10\ 21\ 37\ 48\ \ \underbrace{52\ 63}_{NBSYM}\ ]$$

dove il numero di campioni aggiunti, per ogni parte, è stato deciso in precedenza e in questo caso è pari a $NBSYM = 2$.

2.2.1 Criteri di stop

Nell'algoritmo sono presenti due criteri di stop. Uno per il sifting e un altro per tutto il processo di scomposizione.

Quest'ultimo risulta il più semplice. Se, infatti, la somma del numero dei massimi e del numero dei minimi, prima delle *boundary conditions*, risulta essere minore di tre, allora l'algoritmo termina. Come menzionato in precedenza, questa condizione si verificherà con certezza in un numero finito di iterazioni.

Per quanto riguarda la condizione di stop del processo di sifting, devono essere strettamente rispettate due condizioni:

- il numero dei massimi e dei minimi locali deve differire al massimo di 1.
- la media tra l'*upper* e il *lower envelop* deve essere vicina a zero, secondo alcuni criteri.

Sono stati presentati quindi vari criteri: Huang et al. osservano che limitando il valore della deviazione standard, calcolata tra due sifting successivi come

$$SD = \sum_{t=0}^{T} \left[\frac{|h_{1(k-1)}(t) - h_{1k}(t)|^2}{h_{1(k-1)}^2(t)} \right]$$

a $0.2 \sim 0.3$ può essere una valida e rigorosa limitazione.

Rilling et al. [2] invece introducono due soglie θ_1 e θ_2, con lo scopo di garantire a livello globale piccole fluttuazioni della media tenendo conto intanto di grandi escursioni a livello locale. Introducono quindi l'ampiezza del modo $a(t) = \frac{e_{max}(t) - e_{min}(t)}{2}$ e la *evaluation function* $\sigma(t) = |\frac{m(t)}{a(t)}|$ così il processo è iterato fino alla condizione $\sigma(t) < \theta_1$ per una certa frazione della durata totale $(1 - \alpha)$, mentre $\sigma(t) < \theta_2$ per la parte rimanente (α). I valori assegnati di default sono $\alpha = 0.05$ $\theta_1 = 0.05$ e $\theta_2 = 10$. Nel nostro caso si è preso spunto dall'ultima condizione proposta e la si è adattata come segue: se la cardinalità dell'evento $\sigma(t) > 0.5$ è maggiore della metà della lunghezza del segnale in analisi, oppure

$\sigma(t) > 0.05$ più di 0 volte e il numero di massimi e minimi è superiore a 2 allora procedi, altrimenti fermati.

2.2.2 Risultati

In tutte le simulazioni, sia usando Matlab che il linguaggio C si sono riscontrati risultati positivi e soddisfacenti, nei quali il numero dei modi non era mai superiore a 9, e la cui somma restituiva sempre il segnale iniziale con un errore massimo dell'ordine di 1×10^{-14}, che è confrontabile con l'errore di precisione di Matlab.

2.2.3 Precisazioni

Il metodo proposto è un metodo a posteriori. Esso necessita di una quantità più o meno elevata di campioni per essere efficiente a livello di compressione. Non è possibile quindi, almeno allo stato dell'arte, riuscire a sfruttare l'EMD in tempo reale. Questo si ripercuote sulle possibili applicazioni nelle WSN. Ad esempio non è pensabile di usare tale algoritmo per monitorare sistemi di sicurezza e di allarme, in quanto, potrebbero passare vari istanti prima che il sistema centrale possa accorgersi di una determinata situazione critica. Analogamente, l'algoritmo non può essere usato per le reti multi-hop veicolari, sempre per lo stesso motivo, in quanto queste reti necessitano di trasmissione di informazione in tempo reale. Applicazioni in cui invece l'EMD può essere usato efficientemente sono, ad esempio, il monitoraggio di temperatura o di luminosità di una stanza, oppure il monitoraggio dell'attività elettrica in una rete smart grid. Per concludere, tutto ciò che non necessita di acquisizione ed immediato invio dei dati, può sfruttare questo algoritmo.

2.3 Denoising e selezione dei modi da inviare

Successivamente alla decomposizione del segnale, come presentato precedentemente in relazione a Figura 2.1, si passa alla selezione dei modi da inviare, e quindi al *denoising*. Una volta selezionati i modi da trasmettere, viene calcolata la tolleranza sul singolo modo, che servirà nel calcolo del modello predittivo (sia AR che PLA-LTC).

2.3.1 Denoising

Viene qui calcolata l'autocorrelazione media per ogni modo (funzione *short_time_corr.c*) in modo da escludere quei modi che assomigliano a rumore casuale di tipo impulsivo. Questi modi solitamente sono i primi modi calcolati da EMD e la loro funzione di autocorrelazione è molto piccola, lungo tutti i campioni, e sostanzialmente più piccola di quella dei modi ritenuti significanti. I modi, quindi, sono identificati e scartati, mantenendo quindi $M' \leq M$ modi.

Risultati sperimentali mostrano che solitamente vengono esclusi circa il 30% dei modi calcolati da EMD. Naturalmente questo dipende dalla natura del segnale, e quindi non è da prendere come regola. Ci sono stati casi in cui il segnale di partenza era abbastanza piatto e con un rumore molto limitato, e per il quale si richiedeva l'invio di tutti i modi calcolati.

2.3.2 Calcolo della tolleranza

La tecnica per calcolare la tolleranza è la seguente:

1. si trova il range del segnale $range_{sig}$.
2. la tolleranza sul segnale è $TOL = \gamma \cdot range_{sig}$ dove γ nel nostro caso è stato posto a $\gamma = 0.1$.

3. per ogni $i = 1, 2, \ldots, M'$ si calcola il suo coefficiente di tolleranza come $\alpha_i = \frac{range_{sig}}{range_{IMF}}$.
4. si normalizzano i coefficienti dimodoché la loro somma risulti 1:
$$\forall i : \alpha_i^{norm} = \frac{\alpha_i}{\sum_i \alpha_i}$$
5. le tolleranze per ogni IMF sono: $tol_i = TOL \cdot \alpha_i^{norm}$.

2.4 Adaptive Modeling

Come discusso all'inizio di questo capitolo, successivamente al calcolo dei modi e alla selezioni di tali, vengono inseriti dei modelli adattivi, per rappresentare l'intera sequenza di punti di ogni modo in maniera più efficiente. L'idea è quella di approssimare insiemi di punti consecutivi grazie a funzioni analitiche parametrizzate da un basso numero di coefficienti. Queste funzioni sono mantenute fino a quando l'errore di ricostruzione del segnale in esame rimane all'interno di una certa tolleranza prefissata. In questa sezione si esaminano le tecniche utilizzate per l'approssimazione dei IMFs, che sono di due tipi: (1) autoregressivo (Auto-regressive model, AR), ovvero la costruzione di un modello basato sulla "storia" del segnale e, ad esempio, la sua funzione di autocorrelazione. I modelli AR sono largamente utilizzati per applicazioni ambientali e fisiche, e quindi sono adatti anche per i sistemi di monitoraggio nelle reti di sensori wireless. Quando vengono utilizzati per compressione del segnale, viene ottenuto un modello dai dati in input, il quale viene inviato al ricevitore, il quale prevede i campioni futuri. Il modello è valido finchè non predice il segnale con un errore che eccede la tolleranza massima impostata; (2) lineare, dove viene rappresentato il segnale in ingresso da uno o più segmenti di linea, considerando sempre che l'errore massimo di ricostruzione sia

inferiore ad una determinata tolleranza preimpostata. Il modello, quindi, abbatte il numero di coefficienti da utilizzare per la ricostruzione del segnale, in quanto ogni segmento viene definito da due soli valori (punto iniziale e coefficiente angolare), ma può approssimare anche l'intera sequenza in ingresso.

2.4.1 Auto Regressive Model (AR)

Per ogni modo considerato si calcola il modello AR ad esso associato. Il metodo utilizzato usa quindi un modello AR di ordine p, dividendo il tempo in termini di *prediction cycles*. Sia dunque n l'indice del tempo all inizio del ciclo di predizione. I primi p campioni acquisiti, partendo da $x(n)$, devono essere codificati e trasmessi; verrano poi utilizzati lato ricevitore per inizializzare il predittore. Una volta acquisito il campione $x(n+p+1)$, sono calcolati i p coefficienti del modello $M^{(n,1)} = AR(n,p,1)$, dove n è il punto iniziale della *estimation window* e *p+1* la sua larghezza, ad esempio, i punti considerati sono $\{x(n),\ldots,x(n+p+1)\}$. $M^{(n,1)}$ è quindi utilizzato per predire $\widehat{x}(n+p+1)$, considerando come valori iniziali $\{x(n),\ldots,x(n+p)\}$. Se è verificata la condizione sulla tolleranza, quindi $|\widehat{x}(n+p+1)-x(n+p+1)| < \epsilon$, il modello è temporaneamente considerato valido. Considerando ora il campione successivo, viene calcolato il nuovo modello $M^{(n,2)} = AR(n,p,2)$ sulla finestra $\{x(n),\ldots,x(n+p+2)\}$. Successivamente, $M^{(n,2)}$ è usato per predire $\widehat{x}(n+p+1)$ (un passo avanti) e $\widehat{x}(n+p+2)$ (due passi avanti), con il modello $M^{(n,2)}$ inizializzato sempre con i valori $\{x(n),\ldots,x(n+p)\}$, e i valori predetti sono confrontati con i campioni reali per verificare che $|\widehat{x}(n+p+i) - x(n+p+i)| < \epsilon$ per $i = 1,2$. Il processo è quindi iterato finchè, per qualche valore $k \geq 1, M^{(n,k)}$ non è più in grado di soddisfare le condizioni sulla tolleranza per almeno un campione: ovve-

ro quando $|\hat{x}(n+p+i) - x(n+p+i)| > \epsilon$ per almeno un $i \in \{p+1, \ldots, k\}$. In questo caso, l'ultimo modello valido è $M^{(n,k-1)}$, inizializzato con $\{x(n), \ldots, x(n+p)\}$ e usato per stimare $\{\hat{x}(n+p+1), \ldots, \hat{x}(n+p+k-1)\}$. Comincia quindi un nuovo ciclo di predizione al tempo $n+p+k$ (campione $x(n+p+k)$) e il nuovo modello $M^{(n+p+k,1)}$

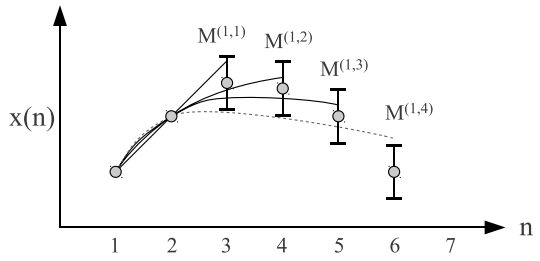


Figura 2.6: Rappresentazione della predizione AR

Il modello di uscita è rappresentato da una matrice a N righe e $1 + 2p$ colonne, dove p è l'ordine del modello. Nella prima colonna è presente l'indice da cui si inizia a ricalcolare il segnale. Nelle collonne $2, 3, \ldots, p+1$ sono presenti i coefficienti del filtro AR, e nelle p successive i valori iniziali da passare in ingresso al filtro. Di norma p è posto essere uguale a 3, solo in alcuni casi uguale a 2, e comunque mai maggiore di 3. Ne risulta quindi che il numero di colonne della matrice è pari a 5 o 7.

In Figura 2.6 si nota il comportamento del modello AR, ogni qualvolta si aggiunge un valore al calcolo, il quale tenta di seguire l'andamento del segnale, rimanendo dentro la tolleranza impostata. Non appena la condizione sulla tolleranza non viene rispettata, si salvano i risultati e si ricomincia con un nuovo modello, partendo dal punto in cui il modello non è ancora stato calcolato (Figure 2.7)

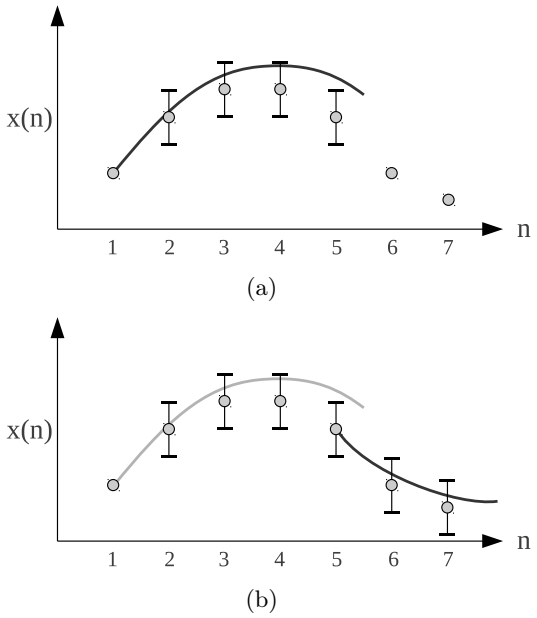


Figura 2.7: Step AR

la funzione *ar* e l'algoritmo *Least Square*

Analizziamo in dettaglio l'algoritmo LS per il calcolo dei coefficienti del filtro di predizione. Questo è contenuto nella funzione *ar*, alla quale viene passato come argomento il pezzo di segnale da predire e l'ordine del filtro.

Vengono innanzitutto inizializzati una matrice C ed un vettore c. C è una matrice quadrata $p \times p$ detta matrice di autocorrelazione e c è un vettore lungo p detto di cross correlazione

$$C_{i,j} = \sum_{t=p}^{N} x(t-i)x(t-j)$$

dove N è la lunghezza del vettore passato per il calcolo

$$c_j = \sum_{t=p}^{N} x(t)x(t-j)$$

Il sistema risultante è

$$Cy = c$$

Tramite il metodo *linsolve* si risolve il sistema utilizzando *Gaussian elimination*. Il vettore di uscita dal metodo *linsolve* è quindi quello dei coefficienti y del filtro ($y = C^{-1}c$) che, come anticipato, viene utilizzato per verificare che il segnale ricalcolato non differisca da quello originale più della tolleranza.

Naturalmente diminuendo il coefficiente γ nel calcolo della tolleranza, aumenta la precisione di ricostruzione, aumentando però il numero di condizioni di "stop" e quindi di righe della matrice del modello AR e dunque aumentando il numero di coefficienti necessari per ricostruire correttamente l'intero segnale in ingresso.

Ad ogni iterazione, se l'errore sul segnale ricostruito è minore della tolleranza, viene aumentata la lunghezza del segnale passato alla funzione *ar*, aggiungendo un elemento al vettore. Quando la condizione di errore non è più verificata, oltre a salvare i dati fin qui calcolati, si riparte a calcolare il modello del segnale partendo dall'indice ultimo dell'iterazione precedente, incrementato di 1.

la funzione *predict_ar*

La funzione *predict_ar* è, insieme alla funzione *ar*, la più importante nel calcolo del modello AR. Essa è utilizzata sia nell'encoder che nel decoder e ricostruisce il segnale partendo dai coefficienti del filtro e dai valori iniziali. Il segnale è così ricostruito

$$y(i) = y(i) - y(i-j)ar\_coefs(j)$$

dove $j = 1, 2, \ldots, p$ e $i = 1, 2, \ldots, N$ dove N è la lunghezza della porzione di segnale da ricostruire. N è calcolato prendendo il coefficiente d'indice successivo a quello in esame nella prima colonna della matrice del modello AR, e sottraendo il coefficiente in esame. In una prima versione vi era una

seconda colonna di coefficienti che determinavano l'indice al quale la condizione di stop non era più verificata, ma si è deciso di eliminarla perchè il dato era ridondante e comportava un aumento del payload.

Si tiene a precisare che, nel caso peggiore, ovvero quando il modello AR non predice correttamente il segnale, vengono inviati coefficienti posti a zero, e si fa per questo affidamento solo sui valori iniziali del modello.

Si ha quindi, purtroppo, un dispendio di risorse, poiché si utilizzano più valori (es: 7) di quelli effettivamente utili (es: 3). È comunque possibile implementare una semplice funzione che, se si verificasse questo caso, possa considerare solo i valori utili del segnale, senza trasmettere valori nulli.

2.4.2 Piecewise Linear Approximation (PLA)

L'algoritmo Piecewise Linear Aproximation (PLA) è una tecnica di approssimazione molto conosciuta. Si basa sul fatto che, per molte sequenze temporali di misurazioni ambientali quali temperatura ed umidità, l'approssimazione lineare lavora bene su brevi periodi. L'idea è quella di usare una sequenza di segmenti per rappresentare la serie $x(n)$ in input con un errore di approssimazione contenuto (vedi Figura 2.8). Inoltre, siccome un segmento di linea può essere determinato da due soli punti, PLA presenta un efficiente rappresentazione della serie temporale in termini di memoria e requisiti di trasmissione. Per la ricostruzione del segnale al ricevitore, al generico istante n, gli istanti di osservazione sono approssimati sulla proiezione verticale del campione attuale sul corrispondente segmento di linea (puntini bianchi in figura). Il segnale ricostruito verrà indicato come $\widehat{x}(n)$. L'errore introdotto è la distanza tra il campione attuale (puntini neri) e il segmento lungo questa proiezione verticale $|\widehat{x}(n) - x(n)|$. Gli algoritmi PLA usano un *least squares fitting* standard per calcolare

i segmenti approssimanti. Spesso è introdotta un'ulteriore semplificazione per ridurre la complessità computazionale, che consiste nel forzare i punti esterni di ogni segmento in modo che diventino punti della serie. Questo rende il least squares fitting non necessario in quanto i segmenti sono pienamente identificati dai punti estremi di $x(n)$ nella finestra temporale considerata. Seguendo questa semplice idea sono stati proposti molti metodi in letteratura. Verrà qui presentato quello più significativo (LTC), da sostituire al calcolo del modello AR come compressione del segnale.

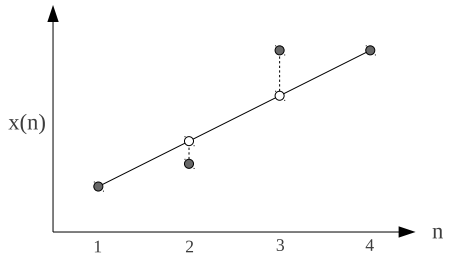


Figura 2.8: Approssimazione di una segnale $x(n)$ con un segmento

Lightweigth Temporal Compression (LTC)

L'algoritmo LTC proposto in [7] è una semplice tecnica PLA a bassa complessità. Nello specifico, siano $x(n)$ i punti di una sequenza temporale con $n = 1, 2, \ldots$. L'algoritmo comincia con $n = 1$ e fissa il primo punto del segmento approssimante a $x(1)$. Il secondo punto $x(2)$ è trasformato in un segmento verticale che determina l'insieme di tutti le linee $\Omega_{1,2}$ accettabili che hanno come punto di partenza $x(1)$. Questo segmento verticale è centrato a $x(2)$ e copre tutti i valori x che soddisfano una tolleranza massima di $\epsilon \geq 0$, ad esempio, $x(2) - \epsilon \leq x \leq x(2) + \epsilon$ Figura 2.9(a). L'insieme delle linee accettabili per $n = 3$, $\Omega_{1,2,3}$, è ottenuto dall'intersezione tra $\Omega_{1,2}$ e l'insieme delle linee con punto di partenza $x(1)$ che sono accettabili per $x(3)$, Figura 2.9(b). Se $x(3)$

cade in $\Omega_{1,2,3}$, l'algoritmo continua con il punto successivo $x(4)$ e il nuovo insieme di linee accettabili $\Omega_{1,2,3,4}$ è ottenuto come intersezione di $\Omega_{1,2,3}$ e l'insieme di linee che partono da $x(1)$ e che sono accettabili per $x(4)$. La procedura è iterata aggiungendo ad ogni passo un punto finchè, ad un dato istante s, $x(s)$ non è contenuto in $\Omega_{1,2,\ldots,s}$. L'algoritmo, quindi, imposta $x(1)$ e $x(s-1)$, rispettivamente, come il punto di partenza e il punto di arrivo del segmento di approssimazione per $n = 1, 2, \ldots, s-1$ e riparte con $x(s)$ come primo punto del prossimo segmento. In Figura 2.9(c) è riportato l'esempio per $s = 4$.

Una caratteristica interessante di questo metodo è la sua adattività: per ogni nuovo campione il segmento approssimante è aggiornato e la tolleranza sull'errore massimo è controllata lungo tutta la linea. Quando l'inclusione di un nuovo campione non soddisfa la massima tolleranza permessa, l'algoritmo comincia a calcolare un nuovo segmento. Quindi si autoadatta alle caratteristiche di $x(n)$. Un'altra caratteristica interessante è che ogni segmento può essere inviato direttamente al decoder, senza avere memorizzato alcun valore passato di $x(n)$.

2.5 Risultati simulativi

Nelle prove effettuate si nota come il segnale ricostruito segue abbastanza bene l'andamento del segnale originale (Figura 2.10), salvo discostarsi in certi punti a causa del predittore AR che si discosta dall'andamento del singolo modo (Figura 2.11). Tuttavia si può osservare come gli errori tra segnale originale e segnale ricostruito soddisfino in pieno le condizione sulla tolleranza impostata, infatti in (Figura 2.12) si è riportato il calcolo della differenza tra il segnale ricostruito e il segnale originale, riferita al segnale di Figura 2.10. Più

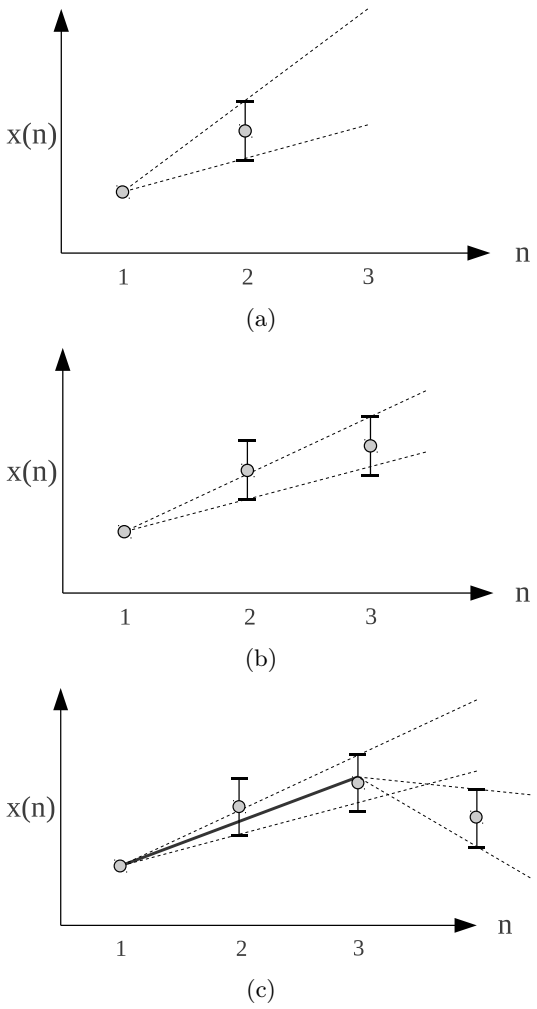


Figura 2.9: Esempio di LTC

in dettaglio, in Figura 2.13, dove vengono riportati i modi selezionati per il segnale di Figura 2.10, si osserva che ogni modo sta perfettamente dentro la soglia di tolleranza impostata. Di facile intuizione è capire quando il modello (in questo caso AR) viene reinizializzato; prendendo come esempio il quarto modo, si osserva un andamento curvilineo dell'errore, il quale si discosta dal valore nullo, e successivamente un repentino ritorno a zero: questo è l'indice della reinizializzazione del modello, il quale non segue più l'andamento del segnale originale (in questo caso del quarto IMF), e eccede la tolleranza.

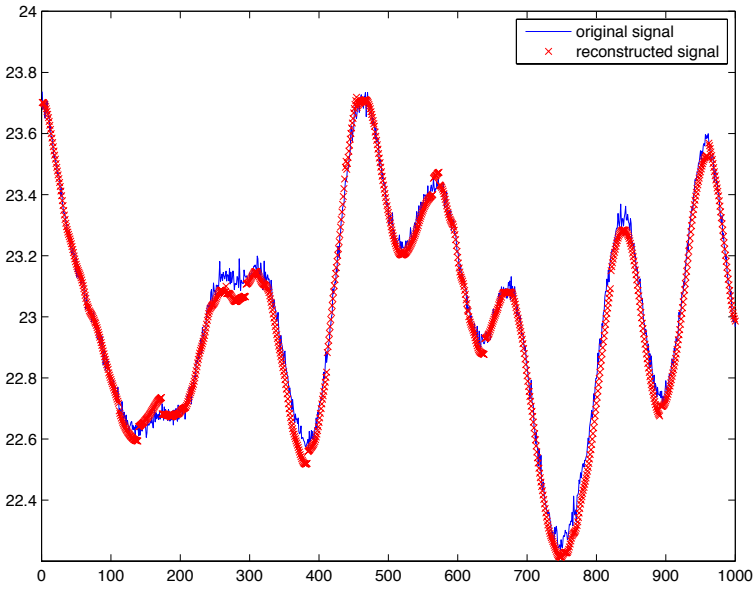


Figura 2.10: Segnale ricostruito mediante modello AR

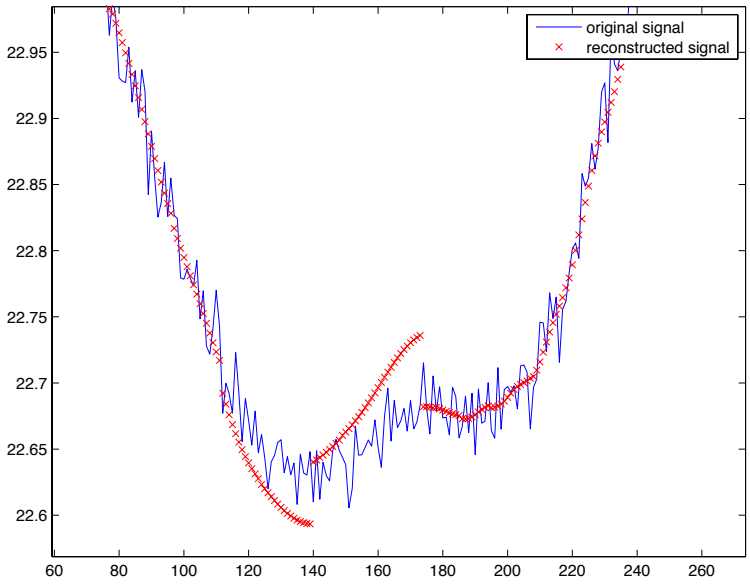


Figura 2.11: Particolare del segnale ricostruito

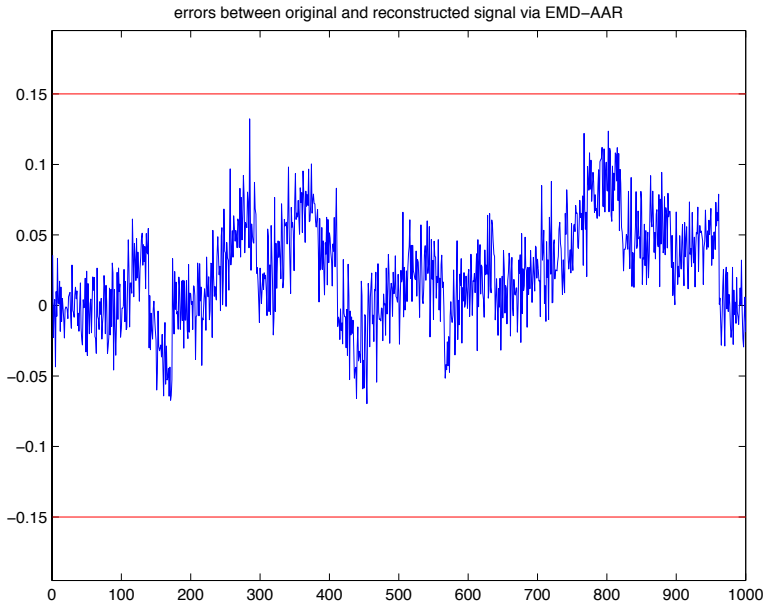


Figura 2.12: Errore tra il segnale originale e il segnale ricostruito

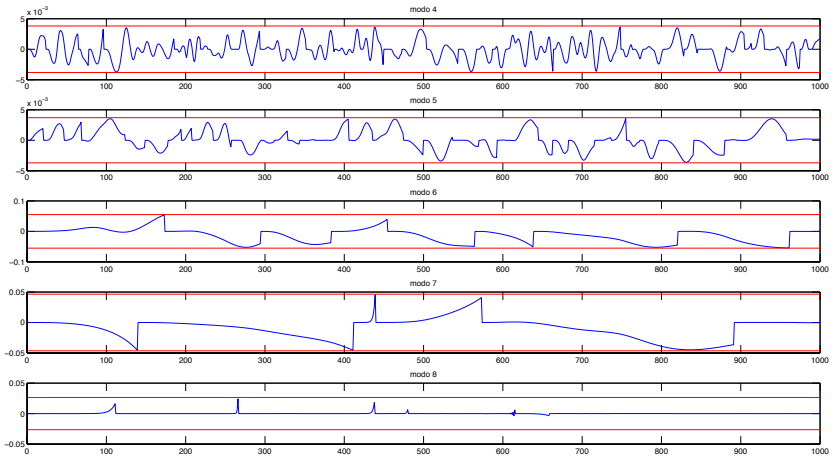


Figura 2.13: Errori nella ricostruzione dei modi con tolleranze

Capitolo 3

Sviluppo del Radio Smart Meter

3.1 Schema Generale

Lo Smart Meter utilizzato per lo svolgimento della tesi è composto dal nodo sensore *Z1* di *Zolertia* (Figura 3.1)[12]. Lo *Z1* è un modulo a basso consumo per reti di sensori che funge da piattaforma generale di sviluppo per WSN, ricerca ecc. Si tratta di una piattaforma compatibile con la famiglia dei nodi Tmote, con diversi miglioramenti che offrono prestazioni di circa due volte più performanti sotto diversi aspetti. Lo *Z1* è compatibile con lo standard IEEE 802.15.4 e protocolli *ZigBee*[13][14]. Viene fornito con supporto per alcuni dei sistemi operativi open source più usati dalla comunità delle WSN, come *TinyOS*. Gli stack di rete supportati includono *6LoWPAN* (mediante BLIP in TinyOS), Texas Instruments 'SimpliciTI e Z-Stack (fino a Zigbee 2006).

Z1 è dotato di una seconda generazione di microcontrollore MSP430F2617 a bassa potenza, che dispone di un potente 16-bit RISC CPU con velocità di clock di 16 MHz, una built-in Clock factory Calibration, 8KB di RAM e una memoria Flash 92KB. Comprende anche il ben noto trasmettitore CC2420, compatibile con IEEE 802.15.4, che funziona

a 2,4 GHz con un data rate effettivo di 250Kbps. La scelta dell'hardware per lo Z1 garantisce la massima robustezza con un basso consumo energetico. Monta inoltre una serie di sensori di vario tipo, come temperatura (TMP102) e un accelerometro (ADXL345), e supporta fino a quattro sensori esterni.

La programmazione avviene tramite USB o tramite adattatore seriale JTAG.

La scheda può essere alimentata in molti modi: Battery Pack (2xAA o 2xAAA), Cell Coin (fino a 3,6 V), via USB o direttamente collegato con due fili arrivanti da una fonte di alimentazione.

Figura 3.1: Nodo sensore Z1 di Zolertia

3.1.1 Scheda MSP430

La scheda programmabile MSP430 fa parte della famiglia dei microcontrollori per mixed-signal di Texas Instruments. Costruita su una CPU a 16-bit, l'MSP430 è progettata per bassi costi, nello specifico, per applicazioni embedded con un basso consumo di energia.

Il MSP430 può essere utilizzato per dispositivi embedded a bassa potenza. La corrente elettrica assorbita in modalità

di riposo può essere inferiore a 1 μA. La velocità massima della CPU è di 25 MHz, ma può essere diminuita per un minor consumo energetico. Il MSP430 utilizza anche sei diverse modalità a bassa potenza, in grado di disattivare i clock e la CPU quando non necessario. Questo permette al MSP430 di entrare in modalità sleep, mentre le sue periferiche continuano a lavorare senza la necessità di un processore, che comporterebbe un dispendio di energia inutile. Inoltre, il MSP430 è in grado di passare da modalità sleep ad essere operativo in un tempo inferiore a $1\mu s$, permettendo al microcontrollore di rimanere in modalità di riposo più a lungo, riducendo al minimo il consumo medio di corrente. Si noti che MHz non è equivalente a milioni di istruzioni al secondo (MIPS), e diverse architetture possono ottenere diversi tassi di CPU MIPS a frequenze di clock più basse, che può portare ad un più basso consumo di potenza dinamica per un importo equivalente di elaborazione digitale. Vi sono, tuttavia, limitazioni che ne impediscono il suo uso in sistemi embedded più complessi. Il MSP430 non dispone di un bus di memoria esterna, quindi è limitato a memoria on-chip (fino a 256 KB di memoria Flash e 16 MB RAM), che potrebbe essere troppo piccolo per le applicazioni che richiedono buffer di grandi dimensioni o tabelle di dati.

Nello specifico, la serie MSP430F2xxx è simile alla generazione '1xxx, ma opera ad un livello di potenza più basso, come visto una frequenza di clock di 16 MHz e un clock interno più accurato, che rende possibile operare senza un cristallo interno. Include un Very-Low power Oscillator. Le specifiche della potenza sono:

- $0.1\mu A$ RAM retention
- $0.3\mu A$ Stand-by mode (VLO)
- $0.7\mu A$ real-time clock mode

- $220\mu A$ MIPS active

3.1.2 TinyOS

TinyOS è un sistema operativo component-based libero e open source per reti di sensori wireless (WSN). TinyOS è un sistema operativo embedded scritto nel linguaggio di programmazione nesC come un insieme di cooperating tasks e processi. TinyOS è iniziato come una collaborazione tra l'Università di California, Berkeley in collaborazione con Intel Research and Crossbow Technology , e da allora è cresciuta fino a diventare un consorzio internazionale, l'Alleanza TinyOS.

Le applicazioni per TinyOS sono scritte in nesC, un "dialetto" del linguaggio C ottimizzato per i limiti di memoria delle reti di sensori. I suoi strumenti supplementari sono principalmente sotto forma di Java e shell script front-end. Librerie associate e strumenti, come il compilatore NESC e Atmel AVR binutils toolchain, sono la maggior parte scritti in C. I programmi di TinyOS sono costituiti di componenti software, alcune delle quali presentano astrazioni hardware. I componenti sono collegati tra loro tramite interfacce. TinyOS fornisce interfacce e componenti per astrazioni comuni, quali la comunicazione dei pacchetti, routing, rilevamento, attuazione e lo stoccaggio. TinyOS è completamente non-blocking: ha uno stack. Pertanto, tutte le operazioni I / O che durano più di qualche centinaio di microsecondi sono asincrone e hanno un callback.

Il codice di TinyOS è linkato staticamente con il codice del programma da eseguire, e compilato in un piccolo file binario, utilizzando una GNU toolchain. Le utilities associate sono fornite per completare una piattaforma di sviluppo per lavorare con TinyOS

3.1.3 Trasmettitore CC2420

Il CC2420 è un trasmettitore single-chip a 2,4 GHz compatibile con lo standard IEEE 802.15.4 RF, progettato per basso consumo e applicazioni wireless a bassa tensione. Include un digital direct sequence spread spectrum baseband modem che fornisce uno spreading factor di 9 dB e un data rate effettivo di 250 kbps. Il CC2420 è a basso costo, ed è una soluzione per comunicazione wireless nella banda dei 2,4 GHz ISM senza licenza. Il CC2420 fornisce un ampio supporto hardware per il packet handling, data buffering, burst transmissions, data encryption, data authentication, clear channel assessment, link quality indication e packet timing information.

3.1.4 Sensore di temperatura TMP102

Il TMP102 è un sensore di temperatura con output seriale a due fili disponibile nel pacchetto SOT563. Non richiede componenti esterne, il TMP102 è in grado di leggere temperature ad una risoluzione di 0,0625 ° C. Il TMP102 è ideale per la misurazione di temperatura in una varietà di applicazioni. Il dispositivo è specificato per il funzionamento in un intervallo di temperatura di -40 ° C a +125 ° C. TMP102 è collegato al MSP430 attraverso il USCIB1 utilizzato come I2C.

3.1.5 6LoWPAN

6LoWPAN è l'acronimo di IPv6 per Wireless Personal Area Networks a bassa potenza. 6LoWPAN è inoltre il nome di un gruppo di lavoro nel settore internet della IETF. Il concetto 6LoWPAN nasce dall'idea che i protocolli Internet potrebbero e dovrebbero essere applicati anche ai più piccoli dispositivi, e che dispositivi a bassa potenza con capacità di elaborazione limitate dovrebbero essere in grado di partecipare alla

Internet of Things. Il gruppo 6LoWPAN ha definito meccanismi di compressione e di incapsulamento che permettono a pacchetti scritti secondo il protocollo IPv6 di essere inviati e ricevuti da reti basate su IEEE 802.15.4. IPv4 e IPv6 sono i cavalli di battaglia per la consegna dei dati per reti locali, reti metropolitane, e wide-area network come Internet. Allo stesso modo, i dispositivi IEEE 802.15.4 forniscono la capacità di rilevamento di comunicazione nel settore wireless. Le nature intrinseche delle due reti, però, è diversa.

Il target per la rete IP per la comunicazione radioa bassa potenza sono le applicazioni che necessitano di connettività internet wireless a velocità di trasferimento dati più bassi per i dispositivi con fattore di forma molto limitata. Gli esempi potrebbero includere: applicazioni di automazione e intrattenimento in ambienti casa, ufficio e di fabbrica. I meccanismi di compressione standardizzati in RFC4944[15] possono essere utilizzati per fornire compressione di header di pacchetti IPv6 su tali reti. IPv6 è utilizzato inoltre su Smart Grid, consentendo agli smart meter e altri dispositivi di costruire una rete prima di inviare i dati al sistema di fatturazione con il backbone. Alcune di queste reti utilizzano 802.15.4.

Come tutti i link-layer di mapping IP, 6LoWPAN fornisce un certo numero di funzioni:

- L'adattamento delle dimensioni dei pacchetti delle due reti: IPv6 richiede il Maximum Transmission Unit (MTU) deve essere almeno 1280 Bytes. Al contrario, la dimensione del pacchetto IEEE802.15.4 's standard è 127 byte.

- Risoluzione degli indirizzi: ai nodi IPv6 vengono assegnati indirizzi IP a 128 bit in modo gerarchico, attraverso un prefisso di rete di lunghezza arbitraria. I dispositivi IEEE 802.15.4 possono invece utilizzare uno degli

indirizzi IEEE a 64 bit estesi o, dopo un evento di associazione, indirizzi a 16 bit che sono unici all'interno di una PAN.

- Progetti di dispositivi diversi: i dispositivi IEEE 802.15.4 sono volutamente limitati per ridurre i costi, ridurre il consumo energetico e permettere flessibilità di installazione. D'altra parte, i nodi cablati nel dominio IP non sono vincolati in questo modo, e possono essere più grandi e utilizzare alimentatori rete.

- Attenzione sull'ottimizzazione dei parametri: nodi IPv6 sono orientati al raggiungimento di alte velocità. Algoritmi e protocolli attuati negli strati superiori sono ottimizzati per gestire problemi di rete tipici quali la congestione.

- Livello di adattamento del formato dei pacchetti per interoperabilità: un meccanismo di adattamento per consentire l'interoperabilità tra dominio IPv6 e la 802.15.4 IEEE.

- Affrontare meccanismi di gestione: la gestione degli indirizzi per i dispositivi che comunicano attraverso i due domini dissimili di IPv6 e IEEE 802.15.4 è ingombrante, se non esaustivamente complessa.

- Periferiche e scoperta del servizio: dal momento che i dispositivi IP-enabled possono richiedere la formazione di reti ad hoc, allo stato attuale, i dispositivi vicini e i servizi ospitati da tali dispositivi dovranno essere conosciuti.

Capitolo 4

Risultati

4.1 Risultati sperimentali su segnali sintetici

Prima di elaborare i dati effettivamente campionati dal sensore, si è testato l'algoritmo su segnali sintetici generati da Matlab, i quali sono stati inclusi in un file *header* chiamato *segnale.h*.

4.1.1 Generazione di segnali sintetici stazionari

I segnali sintetici stazionari sono stati generati grazie ad un metodo conosciuto per imporre il momento primo e secondo ad un processo casuale bianco, [8] [9]. L'obiettivo è quello di ottenere una sequenza casuale $x(n)$ con una data media μ_x, varianza σ_x^2 e funzione di autocorrelazione $\rho_x(n)$. La procedure funziona come segue:

1. Una serie gaussiana casuale $G(k)$ con $k = 1, 2, \ldots, N$ viene generata nel dominio della frequenza, dove N è la lunghezza della sequenza $x(n)$ da ottenere. Ogni elemento di $G(k)$ è una variabile aleatoria gaussiana indipendente con media $\mu_G = 0$ e varianza $\sigma_G^2 = 1$.
2. Viene calcolata la trasformata discreta di Fourier (DFT) della funzione di autocorrelazione $\rho_x(n)$, $S_x(k) = F[\rho_x(n)]$, dove $F[\cdot]$ è l'operatore della trasformata.

3. Si calcola il prodotto interno $X(k) = G(k) \circ S_x(k)^{\frac{1}{2}}$.

4. La sequenza correlata è ottenuta come $F^{-1}[X(k)]$.

La procedura è equivalente al filtraggio di un processo casuale bianco mediante filtro lineare tempo-invariante, la cui funzione di trasferimento è $F^{-1}[S_x(k)^{\frac{1}{2}}]$. La stabilità del procedimento è assicurata da una scelta conveniente della funzione di correlazione, la quale deve essere quadrato sommabile. Per questo scopo si è considerata una funzione di correlazione gaussiana $\rho_x(n) = exp\{-an^2\}$, dove a è scelto in modo da avere la lunghezza di correlazione $n*$ come:

$$a = -\frac{\log(\delta)}{(n*)^2}$$

Senza perdita di generalità, sono stati generati segnali sintetici con $\mu_x = 0$ e $\sigma_x^2 = 1$. Difatti, applicando un offset al segnale generato e un fattore di scala, non cambia la correlazione risultante. Questo modello è generalmente un buon modello per segnali fisici.

In più, per emulare fedelmente il comportamento di un segnale reale, si somma al segnale appena ottenuto del rumore, così da imitare le perturbazioni casuali che affliggono il segnale, dovute alla precisione limitata del sensore e alle fluttuazioni del fenomeno fisico in esame. Questo rumore è modellato come Gaussiano bianco a media nulla con varianza σ_{noise}^2.

4.1.2 Simulazioni su segnali sintetici

L'algoritmo per la generazione dei segnali sintetici è stato utilizzato per la simulazione delle performance in termini di *compression ratio* e di *energy consumption*. Per i riferimenti al consumo energetico in termini di $Joule/sec$ si è fatto uso dei datasheet relativi al MSP430 e al CC2420[17][18]. Per i

risultati sperimentali sono stati generati segnali sintetici con lunghezza di correlazione n^* con $n^*= 10, 20, 40, \ldots, 500$ dove, dopo 40, n^* varia a passi di 20. Si considera una serie temporale di $N = 500$ campioni, estratta da una serie più lunga per evitare problemi dovuti alla generazione del segnale. Al segnale generato è stato aggiunto un rumore Gaussiano bianco con varianza $\sigma^2_{noise} = 0.0016$. Per l'accuratezza di ricostruzione, la tolleranza sull'errore assoluto è stata impostata a $\epsilon = \theta\Delta_x$, dove $\Delta_x = \max_n x(n) - \min_n x(n)$ è il range massimo della serie $x(n)$ e $\theta = 0.1$ è la tolleranza relativa sull'errore. Ogni punto dei grafici delle Figure 4.1 4.2 4.3 4.4 è ottenuto mediando i risultati di 10000 simulazioni. Per una comparazione più equa, lo stesso segnale in input è stato usato per tutti gli ordini, per ogni simulazione e valore di n^*.

Come si può osservare da Figura 4.1 e Figura 4.3, l'utilizzo di EMD conviene solo da un determinato punto in poi, ovvero quando la curva che si riferisce a EMD+AM scende al di sotto del solo AM. Nei risultati trovati questo punto sta ad una lunghezza di correlazione di circa 100-150 campioni. Questo fenomeno è dovuto all'effetto del filtraggio e denoising di EMD. Inoltre, per quanto riguarda il modello AR, si hanno le stesse performance, considerando ordini del polinomio differenti.

Per quanto riguarda la energy consumption, osservando Figura 4.2 e Figura 4.4, si nota che, come prevedibile, all'aumentare della compression ratio, aumenta l'energia spesa per il calcolo dei modelli di predizione. Inoltre, un risultato importante è che, l'energia totale spesa per l'elaborazione e l'invio dei dati è più grande di diversi ordini di grandezza (circa 8) rispetto all'energia spesa per il solo calcolo del modello da inviare. Questo porta alla conclusione che, in termini di consumo di risorse, sia comunque da preferire un algoritmo

più complesso, ma che porti compressioni maggiori, in quanto queste diminuiscono notevolmente il dispendio energetico. Figura 4.2(b) e Figura 4.4(b) confermano quello che è stato appena enunciato [16].

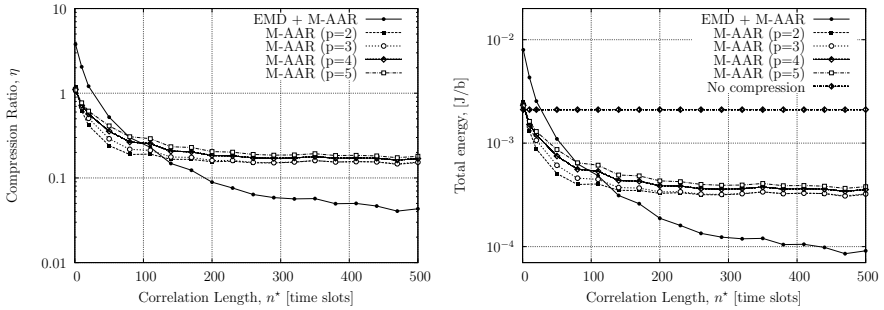


Figura 4.1: Confronto prestazione per la compressione del segnale originale e dei IMFs con modello AR: Compression Ratio vs Correlation Length

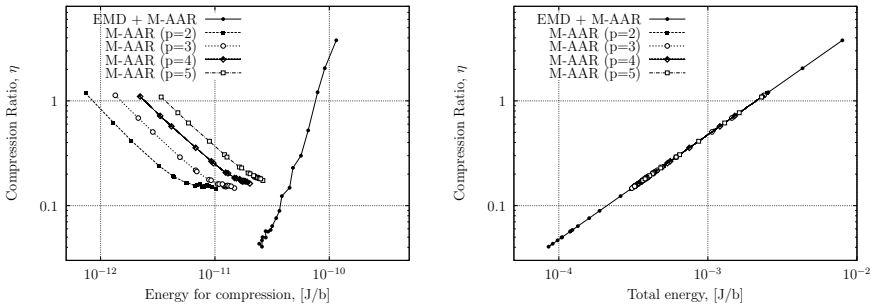
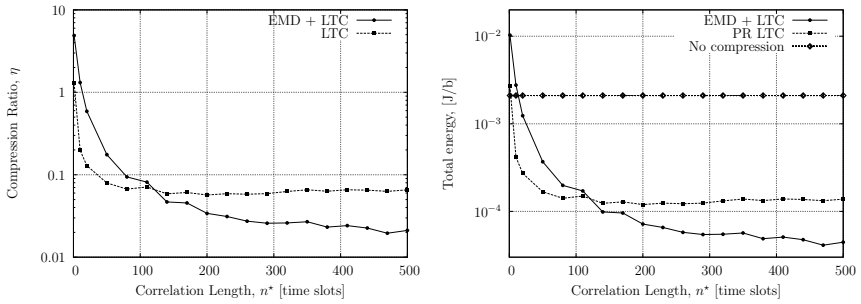


Figura 4.2: Confronto prestazione per la compressione del segnale originale e dei IMFs con modello AR: Compression Ratio vs Complexity

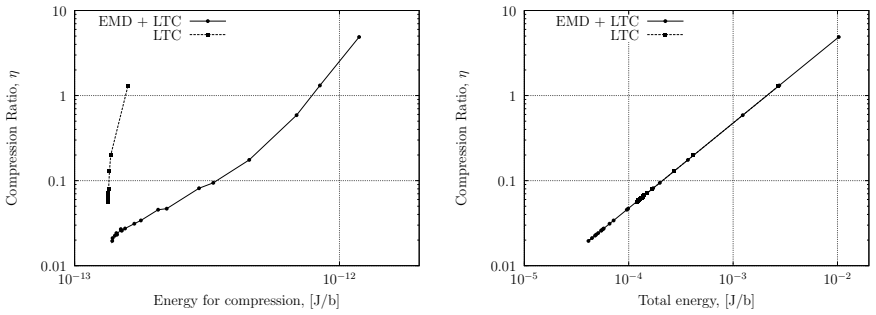
4.1.3 Procedura di test sul sensore

Tramite un debug della scheda, si è verificato che l'algoritmo funziona e termina correttamente. In questo caso si è continuato ad iterare il procedimento di elaborazione dei dati, così da controllare anche possibili errori di overflow di



(a) (b)

Figura 4.3: Confronto prestazione per la compressione del segnale originale e dei IMFs con LTC: Compression Ratio vs Correlation Length



(a) (b)

Figura 4.4: Confronto prestazione per la compressione del segnale originale e dei IMFs con LTC: Compression Ratio vs Complexity

vettori e problemi di memoria. I dati sono stati trasmessi tramite protocolli USB al pc. Lanciando il programma *readvalue.c* si sono acquisiti i dati in ingresso dall'USB e, opportunamente salvati, sono stati utilizzati per la ricostruzione del segnale originale tramite la funzione *aar_ decoder* e la funzione *ltc_ decoder* analizzata nei capitoli precedenti. Il segnale ricostruito (Figura 4.6) rispecchia bene l'andamento del segnale originale , e non ci sono sostanziali differenze tra le prove effettuate con segnali sintetici utilizzando il solo pc e le prove effettuate sulla scheda. Come si può notare dalle Figure 4.5(a) e 4.5(b) l'errore di ricostruzione sul singolo

modo soddisfa alla condizione sulla tolleranza impostata (in rosso) sia per il modello AR che per LTC. In Figura 4.7 poi si riscontra un errore di ricostruzione sull'intero segnale che ancora una volta rimane all'interno del massimo consentito.

4.2 Risultati sperimentali su segnali reali

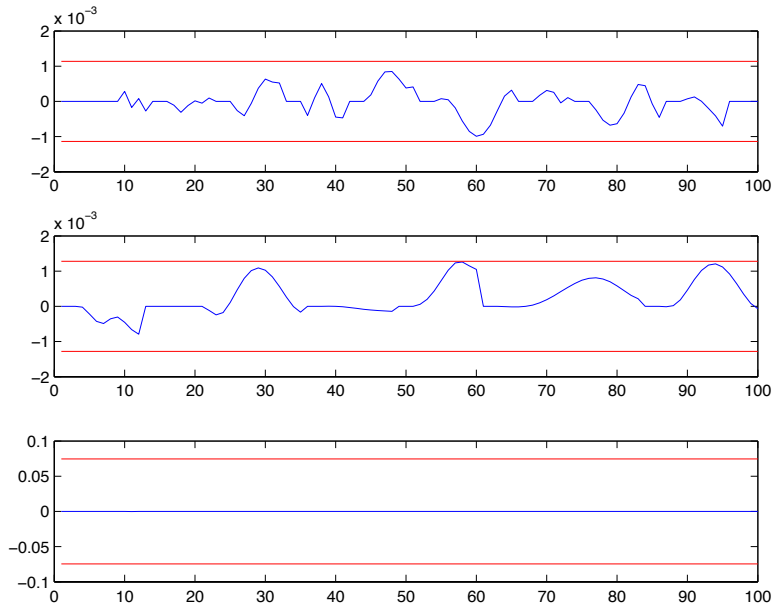
Si sono acquisiti numerosi dati reali di temperatura tramite il sensore Z1, in varie circostanze e luoghi. Si è cercato dunque di differenziare il più possibile le condizioni di misura, in modo da poter osservare comportamenti diversi del segnale. Il sensore è stato infatti posto in luoghi diversi, come appartamenti, laboratori, e outdoor. Ne risultano i segnali in figura 4.8 e 4.9. Come c'era da aspettarsi, l'andamento è molto frastagliato, tuttavia, grazie a EMD, ciò non comporta problemi, in quanto i modi a più alta frequenza non vengono considerati. La ricostruzione è buona (Figure 4.10 e 4.11), anche se ci si poteva aspettare qualcosa in più in termini di precisione. Tuttavia, come negli altri casi, l'errore massimo è sempre minore della tolleranza impostata. I risultati sono quindi soddisfacenti in termini di precisione di ricostruzione, osservando che ciò che comporta errore nella ricostruzione è il rumore, che puntualmente è stato eliminato.

4.2.1 Analisi prestazionale

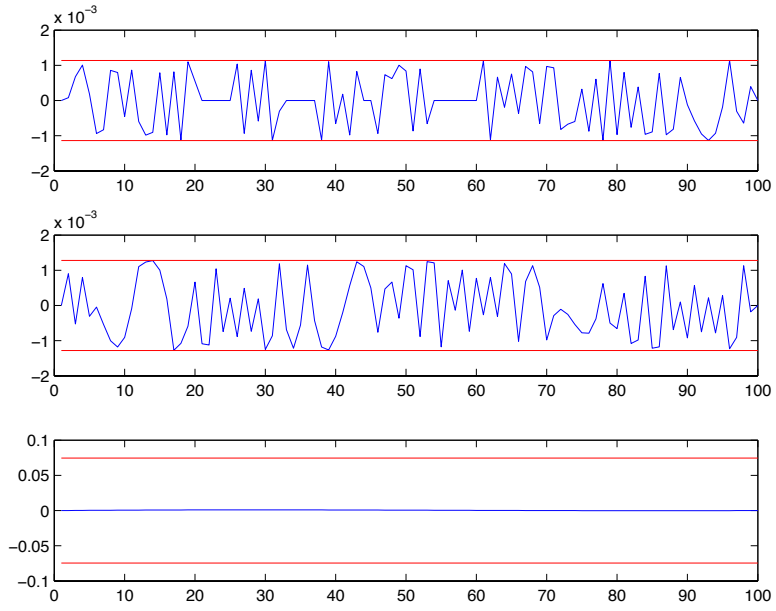
Per verificare quali siano le impostazioni che migliorano la *compression ratio*, si è presa come riferimento una finestra temporale di 1000 campioni. Iterativamente si è divisa la finestra in sotto-finestre da n campioni, con

$$n = 50, 100, 200, 500, 1000$$

e per ognuno di essi si è eseguito EMD seguito sia da AR che da LTC. Si sono poi mediati i valori del numero dei cam-



(a) modello AR



(b) PLA-LTC

Figura 4.5: Errore di ricostruzione sui modi

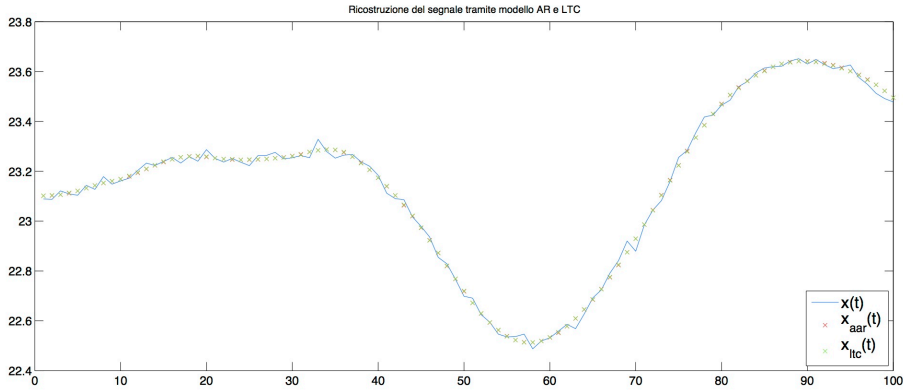


Figura 4.6: Ricostruzione del segnale dopo la compressione-decompressione

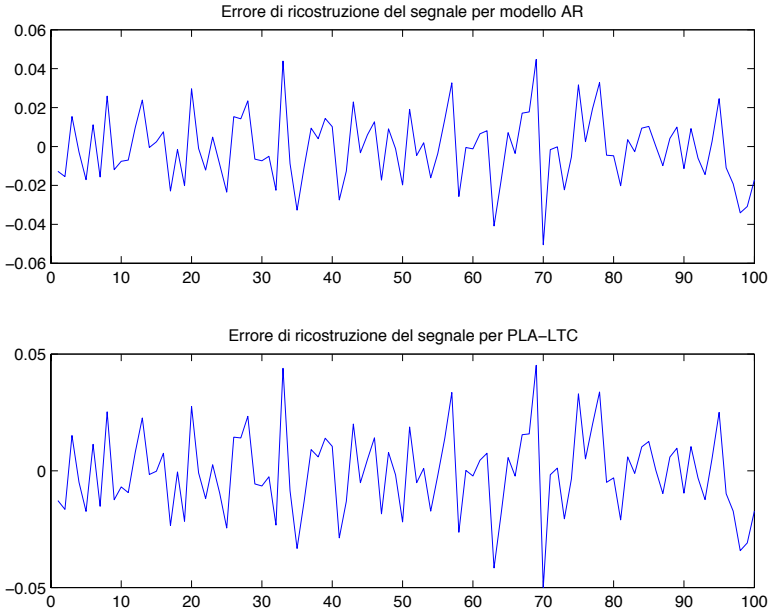


Figura 4.7: Errore di ricostruzione sul segnale

pioni necessari alla codifica per ogni finestra di lunghezza n, in modo da estrarre la *compression ratio* corrispondente alla finestra di 1000 campioni (Figure 4.12(b) e 4.13(b)). Come è possibile osservare in Figura 4.12(a), per il segnale *temp1*, e in Figura 4.13(a) per *temp3*, l'andamento generale è quello che il numero dei campioni da inviare è minore per LTC

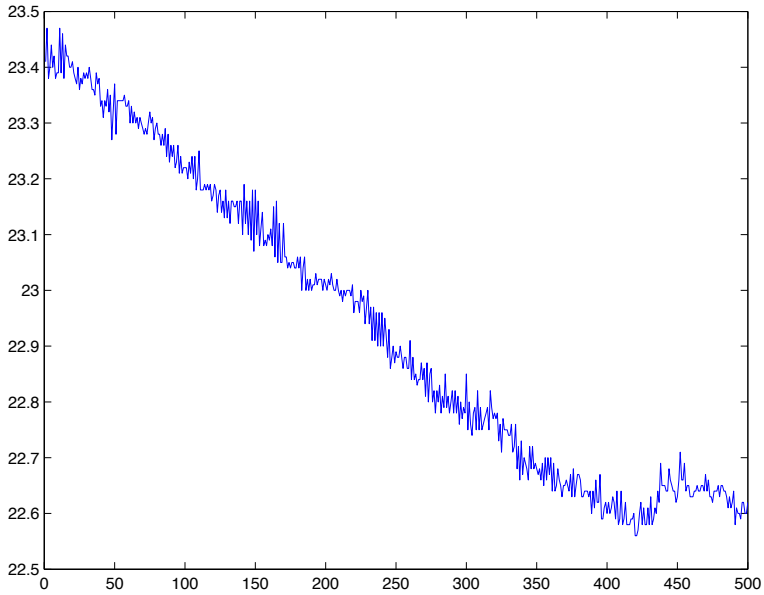


Figura 4.8: Segnale acquisito temp1

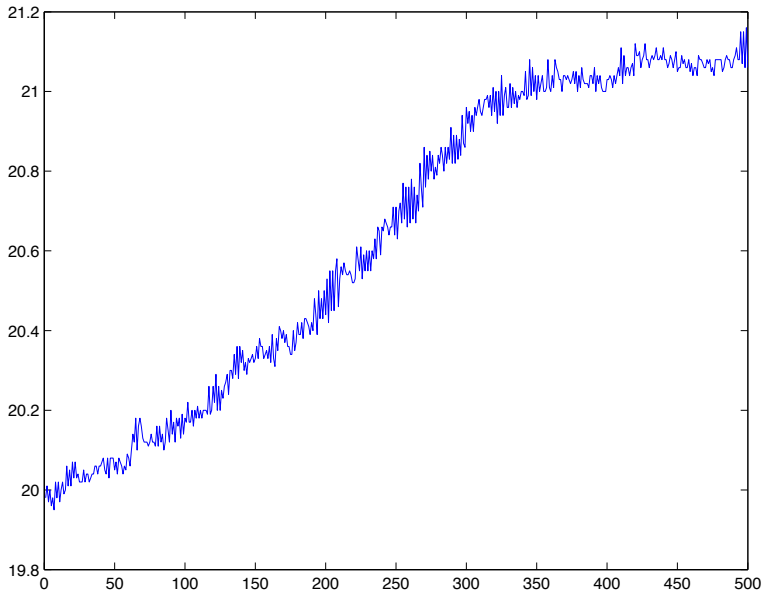


Figura 4.9: Segnale acquisito temp2

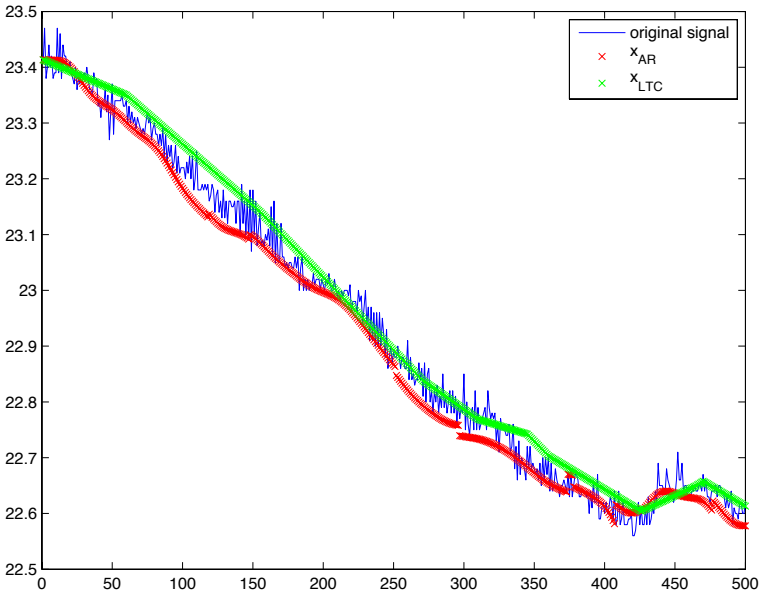


Figura 4.10: Ricostruzione di segnale reale temp1

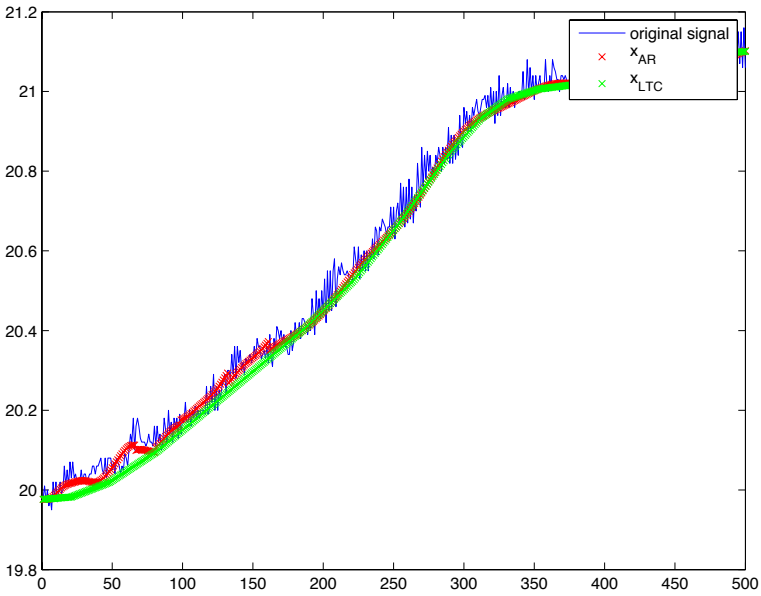


Figura 4.11: Ricostruzione di segnale reale temp2

(in rosso con pallino) rispetto al modello AR (in blu con crocetta), a fronte di un errore quadratico medio pressoché identico (salvo in alcuni casi). In queste figure è infatti riportato il numero di coefficienti (in ordinata) necessari per codificare ogni sottofinestra (20 finestre per $n = 50$, 10 per $n = 100$ ecc), con il corrispondente errore quadratico medio di ricostruzione nella seconda colonna.

Si osserva inoltre che, all'aumentare della larghezza temporale, LTC tende ad aumentare l'efficienza di compressione (Figure 4.12(b) e 4.13(b)), mentre AR non segue un andamento ben definito, questo dovuto alle particolarità di ogni segnale, e quindi di difficile investigazione.

Dalle Figure 4.12(b) e 4.13(b) si evince inoltre che la compressione in termini di coefficienti da inviare via radio può arrivare fino a circa il 5-10% del carico iniziale (senza compressione) per LTC e del 25-35% nel caso del predittore AR.

Si tende quindi a preferire LTC come modello adattivo, vista la notevole differenza in termini di compressione rispetto ad AR, e visto il suo andamento decrescente all'aumentare del numero di campioni considerati, sebbene quest'ultima non sia una regola assoluta, in quanto, all'aumentare della mole di dati considerati, nascono problemi di memoria fisica disponibile nel sensore, che in questo caso è abbastanza limitata (solo 8kB di RAM).

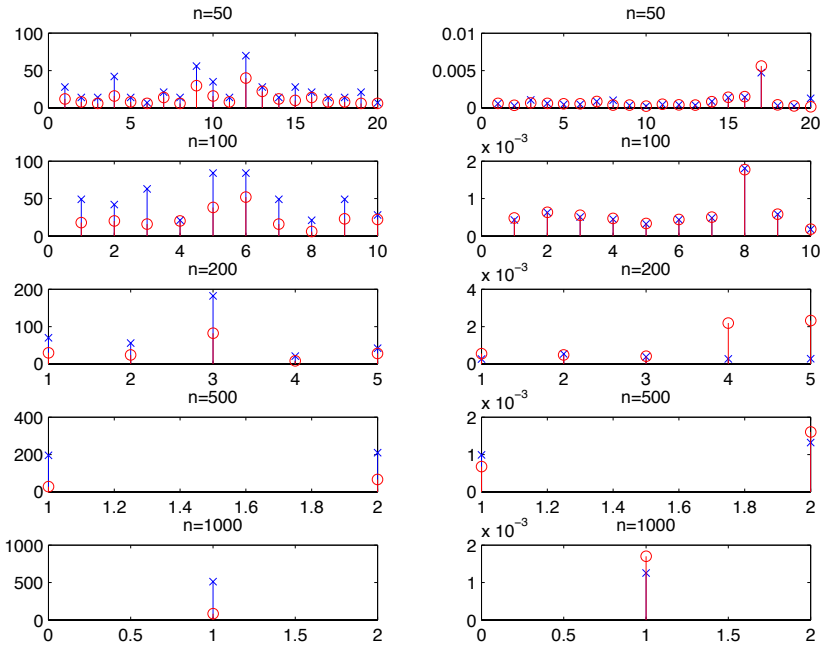
Vengono inoltre presentati due grafici (Figure (4.14) e (4.15)) che permettono di osservare in ascissa l'errore quadratico medio di ricostruzione, ed in ordinata il valore della *compression ratio*. Grazie a questo grafico ed a quelli precedenti, che si riferivano al rate di compressione in base alla lunghezza della finestra, possiamo osservare che una finestra di 200 campioni è il miglior compromesso tra compressione ed errore di ricostruzione, non pesando più di tanto, inoltre,

sulla memoria fisica del sensore.

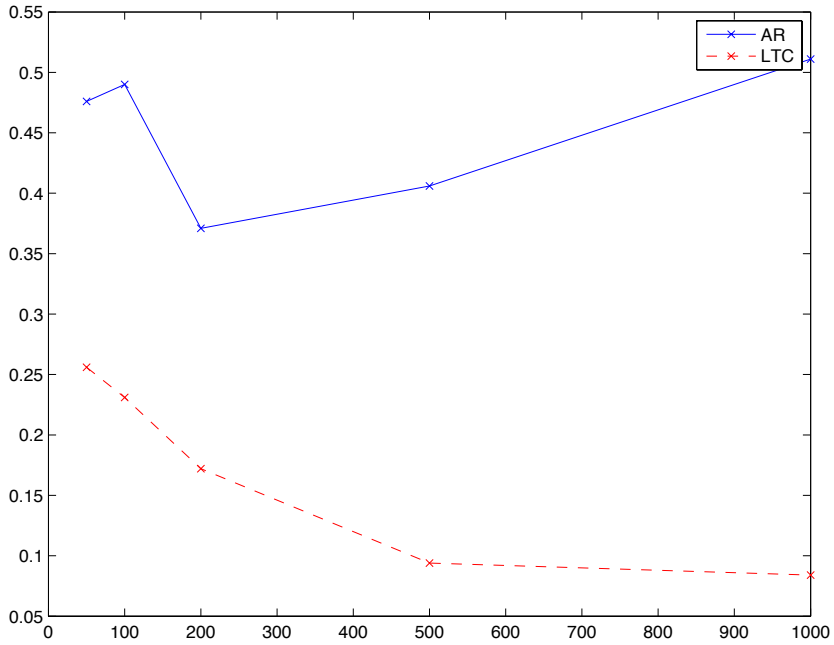
Naturalmente EMD è computazionalmente complesso, in quanto ha bisogno di numerose operazioni di somma e sottrazione, nonché di calcoli di medie e interpolazioni cubiche, che comportano un notevole dispendio di energie e di cicli di clock del microprocessore. Inoltre anche il calcolo del modello AR è computazionalmente complesso e pesante, dovendo ad ogni iterazione invertire matrici e operare su vettori. Questo potrebbe portare a pensare che il compressore in esame non sia efficiente da un punto di vista energetico. Tuttavia, come presentato in [16] e riportato nel caso dei segnali sintetici, l'energia spesa per il calcolo della compressione è di diversi ordini di grandezza più piccola rispetto a quella spesa per l'invio dei dati elaborati. La differenza in termini di spesa di energia si ha solamente sul carico di dati da inviare, e quindi si preferisce la configurazione che massimizza la compressione.

4.2.2 Modelli AR e LTC senza EMD

Per rafforzare la tesi sull'effettivo funzionamento di EMD in termini di efficienza di compressione, si è provato a calcolare sia il modello AR che quello LTC sul segnale originale, ovvero senza che questo venisse scomposto in modi. Ciò che ne risulta è un'ottima compressione (circa 50 campioni da inviare contro i 1000 del segnale originale), ma a fronte di una ricostruzione del segnale pessima, come si può osservare in Figura 4.16. Si è comparato poi il risultato appena ottenuto con la ricostruzione dopo EMD+AM. Come visibile la differenza in termini di ricostruzione è abissale, in quanto con EMD si ha una precisione di gran lunga maggiore, sebbene il numero dei coefficienti da inviare sia salito a circa 350 per AR, mentre solamente a 60 per LTC. Si è visto comunque nel paragrafo 4.1.2 come EMD sia da preferire quando

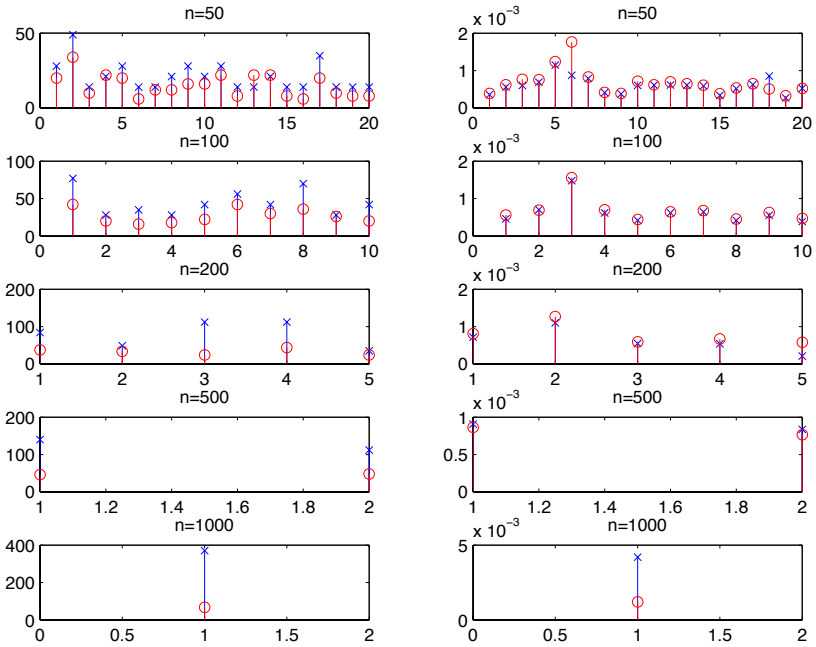


(a)

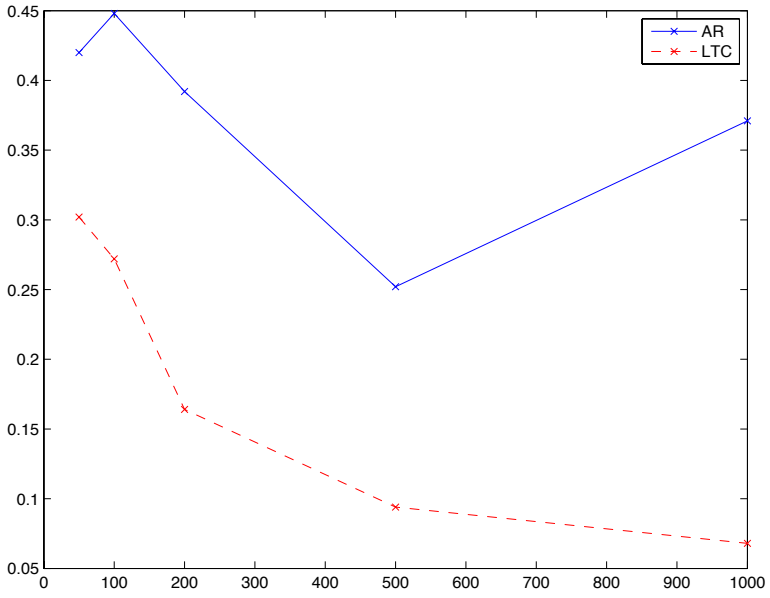


(b)

Figura 4.12: Coefficienti e compressione segnale temp1



(a)



(b)

Figura 4.13: Coefficienti e compressione segnale temp3

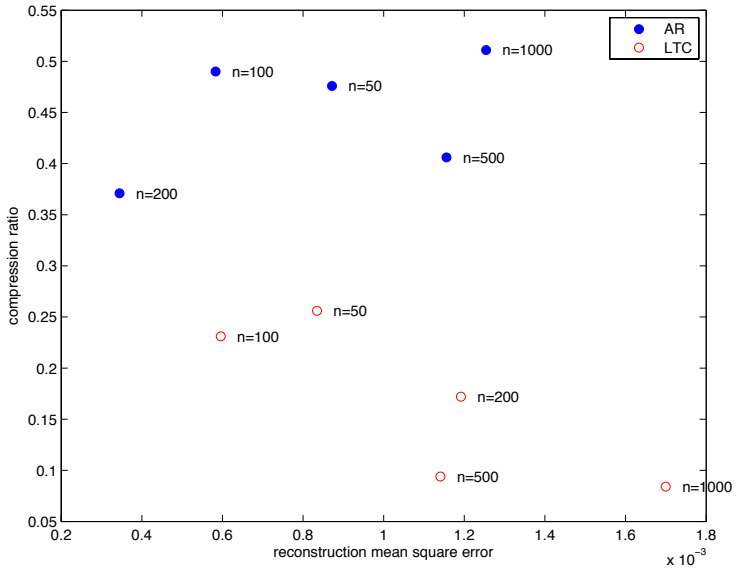


Figura 4.14: errore vs compressione per temp1

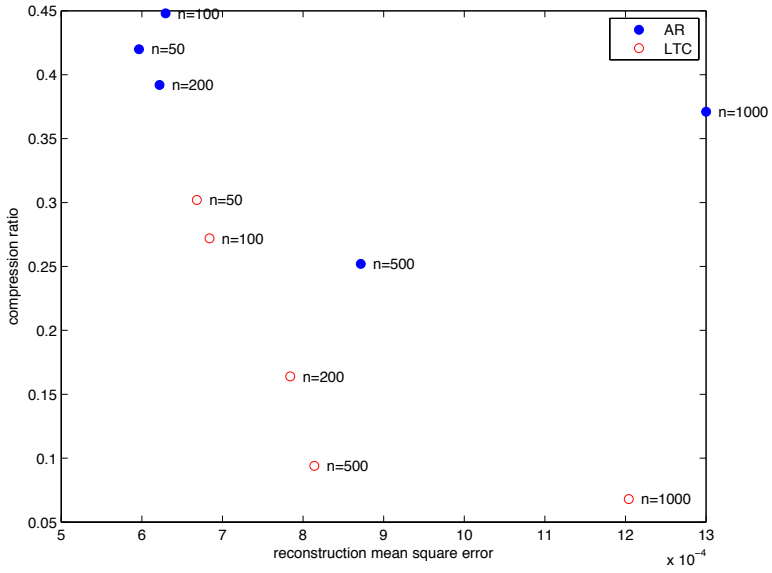


Figura 4.15: errore vs compressione per temp3

la lunghezza di correlazione supera una determinata soglia, mentre non sia da considerare per lunghezze di correlazione

piccole.

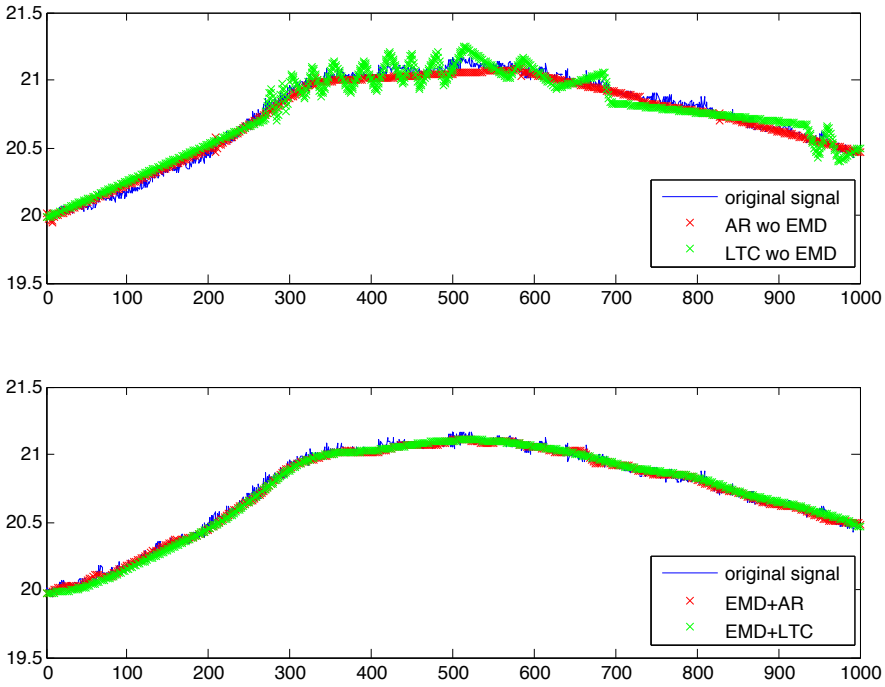


Figura 4.16: Ricostruzione del segnale senza EMD

Capitolo 5

Conclusioni

In questa tesi si è affrontato lo studio di un compressore che può essere utilizzato in diversi ambiti, quali smart grid e rilevazioni ambientali. L'algoritmo può ed è stato implementato nel nodo sensore in maniera semplice e senza incontrare notevoli difficoltà; le limitazioni maggiori si hanno in termini di occupazione di memoria all'interno del sensore. Questo deve avere sufficiente memoria per poter supportare l'acquisizione, l'elaborazione e la successiva trasmissione dei dati. Dai risultati si riscontrano compressioni notevoli, a fronte di un consumo di energia limitato e comunque mai veramente influente sul dispendio di risorse. Il fatto di lavorare su segnali non necessariamente stazionari e lineari apre la strada all'elaborazione di dati di natura diversa, e quindi espande il campo di applicazione delle Wireless Sensor Network. Il maggior pregio di EMD è la sua adattività. L'algoritmo EMD tuttavia non è sempre la scelta migliore in termini di *compression ratio*, ma per gli esempi e i segnali utilizzati e presentati nella tesi il suo utilizzo porta a risultati senza dubbio più performanti di quelli ottenuti senza la sua applicazione. EMD seguito da algoritmi di Adaptive Modeling è quindi una scelta vincente per la maggior parte delle applicazioni delle WSN, e si può proseguire in questa direzione per migliorare sempre di più l'efficienza di compressione al fine

di ridurre al massimo il dispendio energetico e la mole di dati trasmessi via radio, impegnando sempre di meno la rete.

Appendice A

Codici

Sono qui presentate le routine in linguaggio C per l'algoritmo EMD (i codici non presentano commenti per una questione di spazio, sono tutta via presenti nei source code originali)

```
1  /*
2   * aar_ decoder.c
3   *
4   * Created on: Jan 10, 2012
5   * Author: Riccardo Del Re
6   */
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include "operation.h"
10 #include "predict_ar_bmh.h"
11 #include "aar_decoder.h"
12 #include <math.h>
13 #define I(x,y,n) (((x)*(n))+(y))
14
15 void aar_decoder(double model[], short n, char p, double y[],
16                          short length_y,short N) {
17
18         char clmn = 1 + p + p;
19         short j;
20         double xin[p];
21         double phi[p + 1];
22         short length_phi = p + 1;
23         for (j = 0; j < p; j++)
24                 xin[j] = 0;
25         for (j = 0; j < p + 1; j++)
26                 phi[j] = 0;
27
28         char ip = 2;
29         char ix = 2 + p;
30         short i;
31         for (i = 0; i < n; i++) {
32                 short index = model[I(i,0,clmn)];       short size;
33                 if (i < n - 1)
34                         size = model[I(i+1,0,clmn)] - model[I(i,0,clmn)];  else
35                         size = N - model[I(i,0,clmn)] + 1;
36                 for (j = ip - 1; j < ip + p - 1; j++)
37                         phi[j - ip + 2] = model[I(i,j,clmn)];
```

```
38                  for (j = ix - 1; j < ix + p - 1; j++)
39                          xin[j - ix + 1] = model[I(i,j,clmn)];
40                  if (size > p) {
41                          double yp[size];
42
43                          predict_ar_bmh(phi, length_phi, size, xin, yp);
44                          for (j = index - 1; j < index + size - 1; j++)
45                                  y[j] = yp[j - index + 1];
46                  } else {
47                          for (j = index - 1; j < index + size - 1; j++)
48                                  y[j] = xin[j - index + 1];
49                  }
50
51          }
52  }
```

```
1  // * aar_encoder.c
2  // *
3  // * Created on: Dec 20, 2011
4  // * Author: Riccardo Del Re
5  // */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include "operation.h"
10 #include "ar_bmh.h"
11 #include "predict_ar_bmh.h"
12 #include "aar_encoder.h"
13 #include <math.h>
14 #define I(x,y,n) (((x)*(n))+(y))
15
16 short aar_encoder(double x[], short N, char W, char S, char p, char q,
17                   double tol, double model[], double y[], short n_op) {
18
19     char clmn = 1 + p + p;
20         short i = 0;
21         short k = 0;
22         short j;
23
24         double ytmp[N];
25         double ym[N];
26
27         for (j = 0; j < N; j++) {
28         y[j] = 0.0;
29                 ytmp[j] = 0.0;
30                 ym[j] = 0.0;
31         }
32
33         if (W == 0) {
34                 for (j = 0; j < W; j++) {
35                         y[j] = x[j];
36                         ytmp[j] = x[j];
37                 }
38         } else {
39         }
40
41         char ip = 2;
42         char ix = 2 + p;
43
44         i = W + 1;
45
46         short n = 0;
47         unsigned char flag = 0;
48
49         while (i <= (N - p)) {
50                 double xin[p];
51                 short length_xin = 0;
52
53                 n++;
54
55                 for (j = i - 1; j < i + p - 1; j++) {
56                         xin[j - i + 1] = x[j];
57                         length_xin++;
58                 }
59                 for (j = i - 1; j < i + p - 1; j++)
60                         y[j] = xin[j - i + 1];
61                 flag = 0;
```

```
62                  k = i + p;
63
64                  model[I(n-1,0,clmn)] = i;
65                  for (j = ip - 1; j < ip + p - 1; j++)
66                          model[I(n-1,j,clmn)] = 0.0;
67
68                  for (j = ix - 1; j < ix + p - 1; j++)
69                          model[I(n-1,j,clmn)] = xin[j - ix + 1];
70
71                  while (flag == 0 && k <= N) {
72                          short length_w = k - i + 1;
73                          double w[length_w];
74                          for (j = i - 1; j < k; j++)
75                                  w[j - i + 1] = x[j];
76                          double phi[p + 1];
77                          char length_phi = p + 1;
78                          char type = 0;
79
80                          ar(w, length_w, p, phi, length_phi, type);
81
82                          short pred_hor = k - i + 1;
83                          double yp[pred_hor];
84
85                          predict_ar(phi, length_phi, pred_hor, xin, yp);
86
87                          for (j = i - 1; j < k; j++)
88                                  ytmp[j] = yp[j - i + 1];
89                          short length_e = k - i + 1;
90                          double e[length_e];
91                          for (j = i - 1; j < k; j++) {
92                                  e[j - i + 1] = absol(x[j] - ytmp[j]);
93                          }
94
95
96                          double error = findmax(e, length_e);
97                          if (isnan(error)) error = 0.0;
98
99
100                         if (error > tol)
101                                 flag = 1;
102                         else if (isnan(e[length_e-1])) {
103                                 if (k == N)
104                                         flag = 1;
105                                 else
106                                         k++;
107                         } else {
108                                 for (j = i + p; j < k; j++)
109                                         y[j] = ytmp[j];
110                                 model[I(n-1,0,clmn)] = i;
111                                 for (j = ip - 1; j < ip + p - 1; j++)
112                                         model[I(n-1,j,clmn)] = phi[j - ip + 2];
113                                 for (j = ix - 1; j < ix + p - 1; j++)
114                                         model[I(n-1,j,clmn)] = xin[j - ix + 1];
115                                 k++;
116                         }
117                 }
118                 i = k;
119                 n_op++;
120         }
121         if (i <= N) {
122                 n++;
123                 for (j = i; j < N; j++)
```

```
124                             y[j] = x[j];
125
126                     model[I(n-1,0,clmn)] = i;
127                     for (j = ip - 1; j < ip + p - 1; j++)
128                             model[I(n-1,j,clmn)] = 0;
129                     for (j = ix - 1; j < ix + (N - i); j++)
130                             model[I(n-1,j,clmn)] = y[j - ix + i];
131                     n_op++;
132             }
133
134
135             return n;
136     }
```

```c
/*
 * ar.c
 *
 * Created on: Dec 19, 2011
 * Author: Riccardo Del Re
 *
 * Code is adapted from the homonymous matlab code by Davide Zordan
 */
#include <stdio.h>
#include <stdlib.h>
#include "linsolve_bmh.h"
#include "levinson_recursion_linsolve.h"
#include "inversion.h"

#define I(x,y,n) ((x)*(n))+(y)

void ar(double x[], short length_x, char p, double y[], short length_y,
                char type) {
        int i;

        int N = length_x;
        int t, j;

        if (type == 0) {
                double C[9];
                for (j = 0; j < p; j++) {
                        for (i = 0; i < p; i++) {
                                C[I(j,i,p)] = 0.0;
                        }
                }

                double c[3];
                for (j = 0; j < p; j++)
                        c[j] = 0.0;
                for (t = p; t < N; t++) {
                        for (i = 0; i < p; i++) {
                                for (j = 0; j < p; j++) {
                                        C[I(i,j,p)] = C[I(i,j,p)] + x[t - i - 1] * x[t - j - 1];
                                }
                        }
                }

                for (t = p; t < N; t++) {
                        for (j = 0; j < p; j++) {
                                c[j] = c[j] + x[t] * x[t - j - 1];
                        }
                }
                double meno_c[p];
                for (t = 0; t < p; t++) {
                        meno_c[t] = -c[t];
                }

                linsolve_bmh(C, p, meno_c, y);
    else {}
}
```

```
1  /*
2   * Created on: Dec 15, 2011
3   * Author: Riccardo Del Re
4   *
5   * based on a source code by G. Rilling, last modification: 3.2007
6   * gabriel.rilling@ens-lyon.fr
7   *
8   * The code is adapted from a matlab C code and updated concerning on stop criterion
9   * and variables passed as argument
10  *
11  */
12
13 #include <stdlib.h>
14 #include <stdio.h>
15 #include "io.h"
16 #include "extr.h"
17 #include "interp.h"
18 #include "local_mean.h"
19 #include "emdc.h"
20
21 #define I(x,y,n) ((x)*(n))+(y)
22
23 #define STOP_DEFAULT {.threshold = 0.05, .tolerance = 0.05}
24 #define DEFAULT_THRESHOLD 0.05
25 #define DEFAULT_TOLERANCE 0.05
26 #define MAX_ITERATIONS 30
27 #define LIM_GMP 30000
28 #define NBSYM 2
29
30 int stop_sifting(double *, double *, extrema_t *, stop_t *, short, short);
31
32 int emdc(double sig[], short n, double imf[],char max) {
33
34         short i;
35         char stop_status, stop_EMD, ctrl;
36         short nb_imfs,iteration_counter;
37         short N = n;
38         extrema_t ex;
39         envelope_t env;
40         stop_t stop_params;
41         double *x, *t, *y, *m, *moyenne, *a, *r;
42
43         r = (double *) malloc(n * sizeof(double));
44         x = (double *) malloc(n * sizeof(double));
45         t = (double *) malloc(n * sizeof(double));
46         y = (double *) malloc(n * sizeof(double));
47
48         char max_imfs = max;
49
50         for (i = 0; i < n; i++)
51                 t[i] = i;
52         for (i = 0; i < n; i++)
53                 x[i] = sig[i];
54         for (i = 0; i < n; i++)
55                 r[i] = x[i];
56
57         ex = init_extr(n + 2 * NBSYM);
58         m = (double *) malloc(n * sizeof(double));
59         moyenne = (double *) malloc(n * sizeof(double));
60         a = (double *) malloc(n * sizeof(double));
61         env = init_local_mean(n + 2 * NBSYM);
```

```
62
63          nb_imfs = 0;
64          stop_EMD = 0;
65
66          while ((nb_imfs < max_imfs) && !stop_EMD) {
67
68
69                  for (i = 0; i < n; i++)
70                          m[i] = r[i];
71
72                  iteration_counter = 0;
73
74                  stop_status = mean_and_amplitude(t, m, moyenne, a, n, &ex, &env);
75
76                  ctrl = stop_sifting(moyenne, a, &ex, &stop_params, n,
77                                                  iteration_counter);
78
79                  while (!stop_status && ctrl) {
80
81                          for (i = 0; i < n; i++)
82                                  m[i] = m[i] - moyenne[i];
83                          iteration_counter++;
84
85                          stop_status = mean_and_amplitude(t, m, moyenne, a, n, &ex, &env);
86                          ctrl = stop_sifting(moyenne, a, &ex, &stop_params, n,
87                                          iteration_counter);
88
89                  }
90
91                  for (i = 0; i < n; i++)
92                          imf[I(nb_imfs,i,N)] = m[i];
93
94                  nb_imfs++;
95                  for (i = 0; i < n; i++)
96                          r[i] = r[i] - m[i];
97                  stop_EMD = stopmain(&ex);
98          }
99
100         for (i = 0; i < n; i++)
101                 imf[I(nb_imfs,i,N)] = r[i];
102
103         free(r);
104         free(x);
105         free(t);
106         free(y);
107         free(m);
108         free(moyenne);
109         free(a);
110         free_extr(ex);
111         free_local_mean(env);
112
113         return nb_imfs;
114
115
116         }
117
118 int stop_sifting(double *m, double *a, extrema_t *ex, stop_t *sp, short n,
119                 short counter) {
120         short i;
121         double tol, eps;
122         tol = sp->tolerance * n;
123         eps = sp->threshold;
```

```
124
125         double *sx;
126         sx = (double *) malloc(n * sizeof(double));
127         double meansx;
128         for (i = 0; i < n; i++) {
129                 sx[i] = emd_fabs(m[i]) / (emd_fabs(a[i]));
130                 meansx = meansx + sx[i] / n;
131         }
132         double tru = 0;
133         short trutwo = 0;
134         for (i = 0; i < n; i++) {
135                 if (sx[i] > 0.5)
136                         tru++;
137                 if (sx[i] > 0.05)
138                         trutwo++;
139         }
140         tru = tru / n;
141         if ((tru > 0.5 || trutwo != 0) && ((ex->n_min + ex->n_max) > 2))
142                 return 1; //
143         else
144                 return 0;
145
146 }
147
148 int stopmain(extrema_t *ex) {
149
150         if ((ex->n_min + ex->n_max) < 3)
151                 return 1;
152         else
153                 return 0;
154 }
```

```
1  /*
2   *
3   * Created on: Dec 15, 2011
4   * Author: Riccardo Del Re
5   *
6   * original code written by
7   * G. Rilling, last modification: 3.2007
8   * gabriel.rilling@ens-lyon.fr
9   */
10 #include "extr.h"
11 #include <stdlib.h>
12 #include <stdio.h>
13 #define NBSYM 2
14
15 extrema_t init_extr(int n) {
16         extrema_t ex;
17         ex.x_min = (double *) malloc(n * sizeof(double));
18         ex.x_max = (double *) malloc(n * sizeof(double));
19         ex.y_min = (double *) malloc(n * sizeof(double));
20         ex.y_max = (double *) malloc(n * sizeof(double));
21         return ex;
22 }
23
24 void extr(double x[], double y[], int n, extrema_t *ex) {
25         int cour;
26         ex->n_min = 0;
27         ex->n_max = 0;
28
29         for (cour = 1; cour < (n - 1); cour++) {
30                 if (y[cour] <= y[cour - 1] && y[cour] <= y[cour + 1]) {
31         ex->x_min[ex->n_min + NBSYM] = x[cour];
32                 ex->y_min[ex->n_min + NBSYM] = y[cour];
33                 ex->n_min++;
34                 }
35                 if (y[cour] >= y[cour - 1] && y[cour] >= y[cour + 1]) {
36                 ex->x_max[ex->n_max + NBSYM] = x[cour];
37                 ex->y_max[ex->n_max + NBSYM] = y[cour];
38                 ex->n_max++;
39                 }
40         }
41 }
42
43 void boundary_conditions(double x[], double y[], short n, extrema_t *ex) {
44 int cour, nbsym;
45
46 nbsym = NBSYM;
47
48 while (ex->n_min < nbsym + 1 && ex->n_max < nbsym + 1)
49                 nbsym--;
50 if (nbsym < NBSYM) {
51         for (cour = 0; cour < ex->n_max; cour++) {
52         ex->x_max[nbsym + cour] = ex->x_max[NBSYM + cour];
53         ex->y_max[nbsym + cour] = ex->y_max[NBSYM + cour];
54         }
55         for (cour = 0; cour < ex->n_min; cour++) {
56         ex->x_min[nbsym + cour] = ex->x_min[NBSYM + cour];
57         ex->y_min[nbsym + cour] = ex->y_min[NBSYM + cour];
58         }
59 }
60 if (ex->x_max[nbsym] < ex->x_min[nbsym]) {
61 if (y[0] > ex->y_min[nbsym]) {
```

```
62          if (2 * ex->x_max[nbsym] - ex->x_min[2 * nbsym - 1] > x[0]) {
63                  for (cour = 0; cour < nbsym; cour++) {
64                  ex->x_max[cour] = 2 * x[0]
65                                          - ex->x_max[2 * nbsym - 1 - cour];
66                  ex->y_max[cour] = ex->y_max[2 * nbsym - 1 - cour];
67                  ex->x_min[cour] = 2 * x[0]
68                                          - ex->x_min[2 * nbsym - 1 - cour];
69                  ex->y_min[cour] = ex->y_min[2 * nbsym - 1 - cour];
70                  }
71          } else {
72                  for (cour = 0; cour < nbsym; cour++) {
73                  ex->x_max[cour] = 2 * ex->x_max[nbsym] - ex->x_max[2
74                                          * nbsym - cour];
75                  ex->y_max[cour] = ex->y_max[2 * nbsym - cour];
76                  ex->x_min[cour] = 2 * ex->x_max[nbsym] - ex->x_min[2
77                                          * nbsym - 1 - cour];
78                  ex->y_min[cour] = ex->y_min[2 * nbsym - 1 - cour];
79                  }
80          }
81  } else {
82          for (cour = 0; cour < nbsym; cour++) {
83                  ex->x_max[cour] = 2 * x[0] - ex->x_max[2 * nbsym - 1 - cour];
84                  ex->y_max[cour] = ex->y_max[2 * nbsym - 1 - cour];
85          }
86          for (cour = 0; cour < nbsym - 1; cour++) {
87                  ex->x_min[cour] = 2 * x[0] - ex->x_min[2 * nbsym - 2 - cour];
88                  ex->y_min[cour] = ex->y_min[2 * nbsym - 2 - cour];
89          }
90          ex->x_min[nbsym - 1] = x[0];
91          ex->y_min[nbsym - 1] = y[0];
92  }
93   else {
94          if (y[0] < ex->y_max[nbsym]) {
95          if (2 * ex->x_min[nbsym] - ex->x_max[2 * nbsym - 1] > x[0]) {
96                  for (cour = 0; cour < nbsym; cour++) {
97                  ex->x_max[cour] = 2 * x[0]
98                                          - ex->x_max[2 * nbsym - 1 - cour];
99                  ex->y_max[cour] = ex->y_max[2 * nbsym - 1 - cour];
100                 ex->x_min[cour] = 2 * x[0]
101                                         - ex->x_min[2 * nbsym - 1 - cour];
102                 ex->y_min[cour] = ex->y_min[2 * nbsym - 1 - cour];
103                 }
104         } else {
105                 for (cour = 0; cour < nbsym; cour++) {
106                 ex->x_max[cour] = 2 * ex->x_min[nbsym] - ex->x_max[2
107                                         * nbsym - 1 - cour];
108                 ex->y_max[cour] = ex->y_max[2 * nbsym - 1 - cour];
109                 ex->x_min[cour] = 2 * ex->x_min[nbsym] - ex->x_min[2
110                                         * nbsym - cour];
111                 ex->y_min[cour] = ex->y_min[2 * nbsym - cour];
112                 }
113         }
114 } else {
115         for (cour = 0; cour < nbsym; cour++) {
116                 ex->x_min[cour] = 2 * x[0] - ex->x_min[2 * nbsym - 1 - cour];
117                 ex->y_min[cour] = ex->y_min[2 * nbsym - 1 - cour];
118         }
119         for (cour = 0; cour < nbsym - 1; cour++) {
120                 ex->x_max[cour] = 2 * x[0] - ex->x_max[2 * nbsym - 2 - cour];
121                 ex->y_max[cour] = ex->y_max[2 * nbsym - 2 - cour];
122         }
123         ex->x_max[nbsym - 1] = x[0];
```

```
124         ex->y_max[nbsym - 1] = y[0];
125 }
126 }
127 (ex->n_min) += nbsym - 1;
128 (ex->n_max) += nbsym - 1;
129 if (ex->x_max[ex->n_max] < ex->x_min[ex->n_min]) {
130 if (y[n - 1] < ex->y_max[ex->n_max]) {
131         if (2 * ex->x_min[ex->n_min] - ex->x_max[ex->n_max - nbsym + 1]
132                         < x[n - 1]) {
133                 for (cour = 0; cour < nbsym; cour++) {
134                 ex->x_max[ex->n_max + 1 + cour] = 2 * x[n - 1]
135                                         - ex->x_max[ex->n_max - cour];
136                 ex->y_max[ex->n_max + 1 + cour] = ex->y_max[ex->n_max
137                                         - cour];
138                 ex->x_min[ex->n_min + 1 + cour] = 2 * x[n - 1]
139                                         - ex->x_min[ex->n_min - cour];
140                 ex->y_min[ex->n_min + 1 + cour] = ex->y_min[ex->n_min
141                                         - cour];
142                 }
143         } else {
144                 for (cour = 0; cour < nbsym; cour++) {
145                 ex->x_max[ex->n_max + 1 + cour] = 2 * ex->x_min[ex->n_min]
146                                         - ex->x_max[ex->n_max - cour];
147                 ex->y_max[ex->n_max + 1 + cour] = ex->y_max[ex->n_max
148                                         - cour];
149                 ex->x_min[ex->n_min + 1 + cour] = 2 * ex->x_min[ex->n_min]
150                                                 - ex->x_min[ex->n_min - 1 - cour];
151                 ex->y_min[ex->n_min + 1 + cour] = ex->y_min[ex->n_min - 1
152                                                 - cour];
153                         }
154                 }
155         } else {
156                 for (cour = 0; cour < nbsym; cour++) {
157                 ex->x_min[ex->n_min + 1 + cour] = 2 * x[n - 1]
158                                 - ex->x_min[ex->n_min - cour];
159                 ex->y_min[ex->n_min + 1 + cour] =
160                   ex->y_min[ex->n_min - cour];
161                 }
162         for (cour = 0; cour < nbsym - 1; cour++) {
163                         ex->x_max[ex->n_max + 2 + cour] = 2 * x[n - 1]
164                                         - ex->x_max[ex->n_max - cour];
165                 ex->y_max[ex->n_max + 2 + cour] =
166                       ex->y_max[ex->n_max - cour];
167                         }
168                         ex->x_max[ex->n_max + 1] = x[n - 1];
169                         ex->y_max[ex->n_max + 1] = y[n - 1];
170                 }
171         } else {
172                 if (y[n - 1] > ex->y_min[ex->n_min]) {
173                 if (2 * ex->x_max[ex->n_max] -
174             ex->x_min[ex->n_min - nbsym + 1]< x[n - 1]) {
175                         for (cour = 0; cour < nbsym; cour++) {
176                         ex->x_max[ex->n_max + 1 + cour] = 2 * x[n - 1]
177                                                 - ex->x_max[ex->n_max - cour];
178                         ex->y_max[ex->n_max + 1 + cour] = ex->y_max[ex->n_max
179                                                 - cour];
180                         ex->x_min[ex->n_min + 1 + cour] = 2 * x[n - 1]
181                                                 - ex->x_min[ex->n_min - cour];
182                         ex->y_min[ex->n_min + 1 + cour] = ex->y_min[ex->n_min
183                                                 - cour];
184                         }
185                 } else {
```

```
186                         for (cour = 0; cour < nbsym; cour++) {
187                         ex->x_max[ex->n_max + 1 + cour] =
188                         2 * ex->x_max[ex->n_max]
189                                                 - ex->x_max[ex->n_max - 1 - cour];
190                         ex->y_max[ex->n_max + 1 + cour] =
191                            ex->y_max[ex->n_max - 1- cour];
192                         ex->x_min[ex->n_min + 1 + cour] =
193                         2 * ex->x_max[ex->n_max]
194                         - ex->x_min[ex->n_min - cour];
195                         ex->y_min[ex->n_min + 1 + cour] =
196                         ex->y_min[ex->n_min- cour];
197                         }
198                 }
199                 } else {
200                 for (cour = 0; cour < nbsym; cour++) {
201                         ex->x_max[ex->n_max + 1 + cour] = 2 * x[n - 1]
202                                         - ex->x_max[ex->n_max - cour];
203                         ex->y_max[ex->n_max + 1 + cour] =
204                        ex->y_max[ex->n_max - cour];
205                 }
206                 for (cour = 0; cour < nbsym - 1; cour++) {
207                         ex->x_min[ex->n_min + 2 + cour] = 2 * x[n - 1]
208                                         - ex->x_min[ex->n_min - cour];
209                         ex->y_min[ex->n_min + 2 + cour] =
210                       ex->y_min[ex->n_min - cour];
211                         }
212                         ex->x_min[ex->n_min + 1] = x[n - 1];
213                         ex->y_min[ex->n_min + 1] = y[n - 1];
214                 }
215         }
216
217         (ex->n_min) = ex->n_min + nbsym + 1;
218         (ex->n_max) = ex->n_max + nbsym + 1;
219 }
```

```
1  /*
2   * interp.c
3   *
4   * Created on: Dec 14, 2011
5   * Author: Riccardo Del Re
6   */
7
8  /*
9   * interp.c
10  *
11  * Created on: Dec 12, 2011
12  * Author: Riccardo Del Re
13  * Based on C source by: Peter & Nigel,
14  * Design Software,
15  * 42 Gubberley St,
16  * Kenmore, 4069,
17  * Australia.
18  *
19  * Adapted from the text:
20  * Forsythe, G.E., Malcolm, M.A. and Moler, C.B. (1977)
21  * "Computer Methods for Mathematical Computations"
22  * Prentice Hall
23  *
24  * function interpolation computes coefficients
25  * of tridiagonal matrix by function spline and returns
26  * the point to point spline intepolation
27  * of the knot by the function seval and returns it
28  *
29  */
30
31 #include "interp.h"
32
33 int spline(short n, int end1, int end2, double slope1, double slope2, double x[],
34                 double y[], double b[], double c[], double d[])
35
36 {  int nm1, ib, i;
37         double t;
38
39         nm1 = n - 1;
40
41         if (n >= 3) {
42                 d[0] = x[1] - x[0];
43                 c[1] = (y[1] - y[0]) / d[0];
44                 for (i = 1; i < nm1; ++i) {
45                         d[i] = x[i + 1] - x[i];
46                         b[i] = 2.0 * (d[i - 1] + d[i]);
47                         c[i + 1] = (y[i + 1] - y[i]) / d[i];
48                         c[i] = c[i + 1] - c[i];
49                 }
50
51
52                 b[0] = -d[0];
53                 b[nm1] = -d[n - 2];
54                 c[0] = 0.0;
55                 c[nm1] = 0.0;
56                 if (n != 3) {
57                         c[0] = c[2] / (x[3] - x[1]) - c[1] / (x[2] - x[0]);
58                         c[nm1] = c[n - 2] / (x[nm1] - x[n - 3]) - c[n - 3] / (x[n - 2]
59                                         - x[n - 4]);
60                         c[0] = c[0] * d[0] * d[0] / (x[3] - x[0]);
61                         c[nm1] = -c[nm1] * d[n - 2] * d[n - 2] / (x[nm1] - x[n - 4]);
```

```
62                  }
63
64
65                  if (end1 == 1) {
66                          b[0] = 2.0 * (x[1] - x[0]);
67                          c[0] = (y[1] - y[0]) / (x[1] - x[0]) - slope1;
68                  }
69                  if (end2 == 1) {
70                          b[nm1] = 2.0 * (x[nm1] - x[n - 2]);
71                          c[nm1] = slope2 - (y[nm1] - y[n - 2]) / (x[nm1] - x[n - 2]);
72                  }
73
74
75                  for (i = 1; i < n; ++i) {
76                          t = d[i - 1] / b[i - 1];
77                          b[i] = b[i] - t * d[i - 1];
78                          c[i] = c[i] - t * c[i - 1];
79                  }
80
81
82                  c[nm1] = c[nm1] / b[nm1];
83                  for (ib = 0; ib < nm1; ++ib) {
84                          i = n - ib - 2;
85                          c[i] = (c[i] - d[i] * c[i + 1]) / b[i];
86                  }
87
88
89                  b[nm1] = (y[nm1] - y[n - 2]) / d[n - 2] + d[n - 2] * (c[n - 2] + 2.0
90                                  * c[nm1]);
91                  for (i = 0; i < nm1; ++i) {
92                          b[i] = (y[i + 1] - y[i]) / d[i] - d[i] * (c[i + 1] + 2.0 * c[i]);
93                          d[i] = (c[i + 1] - c[i]) / d[i];
94                          c[i] = 3.0 * c[i];
95                  }
96                  c[nm1] = 3.0 * c[nm1];
97                  d[nm1] = d[n - 2];
98
99          }
100         else
101         {
102                 b[0] = (y[1] - y[0]) / (x[1] - x[0]);
103                 c[0] = 0.0;
104                 d[0] = 0.0;
105                 b[1] = b[0];
106                 c[1] = 0.0;
107                 d[1] = 0.0;
108         }
109
110         return 0;
111 }
112
113 double seval(short n, double u, double x[], double y[], double b[], double c[],
114                 double d[], short *last)
115
116 {
117         int i, j, k;
118         double w;
119
120         i = *last;
121         if (i >= n - 1)
122                 i = 0;
123         if (i < 0)
```

```
124                     i = 0;
125
126             if ((x[i] > u) || (x[i + 1] < u)) {   i = 0;
127                     j = n;
128                     do {
129                             k = (i + j) / 2;
130                             if (u < x[k])
131                                     j = k;
132                             if (u >= x[k])
133                                     i = k;
134                     }
135                     while (j > i + 1);
136             }
137             *last = i;
138
139
140             w = u - x[i];
141             w = y[i] + w * (b[i] + w * (c[i] + w * d[i]));
142             return (w);
143     }
144
145     void interpolation(double yy[], double x[], double y[], short n, double xx[],
146                     short nn) {
147
148             short last;
149             double u;
150             double b[n - 1];
151             double c[n - 1];
152             double d[n - 1];
153
154
155             spline(n, 0, 0, 0, 0, x, y, b, c, d);
156
157             short i;
158             last = 0;
159
160             for (i = 0; i < nn; i++) {
161                     u = xx[i];
162                     if (xx[i] >= x[last])
163                             last++;
164                     yy[i] = seval(n, u, x, y, b, c, d, &last);
165             }
166     }
```

```c
/*
 * linsolve.c
 *
 * Created on: Dec 19, 2011
 * Author: Riccardo Del Re
 *
 * Code is adapted from the homonymous matlab code by Davide Zordan
 */
#include <stdio.h>
#include <stdlib.h>
#include "linsolve.h"
#include "operation.h"

#define I(x,y,n) (x)*(n)+(y)
#define PRECISION 0.0000000000000000001


void swap(double mat[], short n, short k, char maxi) {
	double *temp;
	int i;
	temp = (double *) malloc(n * sizeof(double));
	for (i = 0; i < n; i++) {
		temp[i] = mat[I(k,i,n)];
		mat[I(k,i,n)] = mat[I(maxi,i,n)];
		mat[I(maxi,i,n)] = temp[i];
	}
	free(temp);
}

void swaparray(double mat[], short k, short maxi) {
	double temp;
	temp = mat[k];
	mat[k] = mat[maxi];
	mat[maxi] = temp;

}

void linsolve(double mat[], char n, double vec[], double x[]) {
	short i, j, k;
	char maxi;
	double max, pivot, q,h;


	i = 1;
	j = 1;
	k = 1;

	for (k = 0; k < n - 1; k++) {
		max = absol(mat[I(k,k,n)]);
		maxi = k;
		for (i = k; i < n; i++) {
			h = absol(mat[I(i,k,n)]);
			if (h > max) {
				max = h;
				maxi = i;
			}
		}
		if (maxi != k) {
			swap(mat, n, k, maxi);
			swaparray(vec, k, maxi);
		}
```

```
62
63                  pivot = mat[I(k,k,n)];
64                  if (absol(pivot)<PRECISION) pivot =0.0;
65
66                  if (k+1==n){
67                          i = k+1;
68                          q = −mat[I(i,k,n)] / pivot;
69
70                          mat[I(i,k,n)] = 0.0;
71
72
73                          j=k+1;
74                          mat[I(i,j,n)] = mat[I(i,j,n)] + q ∗ mat[I(k,j,n)];
75                          if (absol(mat[I(i,j,n)])<PRECISION) mat[I(i,j,n)] =0.0;
76
77                          vec[i] = vec[i] + q ∗ vec[k];
78                          if (absol(vec[i])<PRECISION) vec[i] =0.0;
79                  }
80                  else{
81                  for (i = k + 1; i < n; i++) {
82
83
84                          q = −mat[I(i,k,n)] / pivot;
85                          mat[I(i,k,n)] = 0.0;
86
87                          for (j = k + 1; j < n; j++) {
88                                  mat[I(i,j,n)] = mat[I(i,j,n)] + q ∗ mat[I(k,j,n)];
89                                  if (absol(mat[I(i,j,n)])<PRECISION) mat[I(i,j,n)] =0.0;
90                          }
91                          vec[i] = vec[i] + q ∗ vec[k];
92                          if (absol(vec[i])<PRECISION) vec[i] =0.0;
93
94                  }
95                  }
96          }
97
98
99          vec[n − 1] = vec[n − 1] / mat[I(n−1,n−1,n)];
100         for (i = n − 2; i >= 0; i−−) {
101                 for (j = n − 1; j > i; j−−) {
102                         vec[i] = vec[i] − mat[I(i,j,n)] ∗ vec[j];
103                 }
104                 vec[i] = vec[i] / mat[I(i,i,n)];
105         }
106
107         x[0] = 1;
108         for (i = 1; i <= n; i++) x[i] = vec[i−1];
109 }
```

```
1  /*
2   * local_mean.c
3   *
4   * Created on: Dec 15, 2011
5   * Author: Riccardo Del Re
6   *
7   * original code by G. Rilling, last modification: 3.2007
8   * gabriel.rilling@ens-lyon.fr
9   *
10  * updated concerning function "interpolation" that didn't work in the previous version
11  * now spline interpolation is the same as the matlab "spline" function
12  */
13
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include "local_mean.h"
17 #include "interp.h"
18 #include "newinterpolation.h"
19 #define LIM_GMP 30000
20
21 envelope_t init_local_mean(int n) {
22         envelope_t env;
23         env.e_min = (double*) malloc(n * sizeof(double));
24         env.e_max = (double*) malloc(n * sizeof(double));
25         env.tmp1 = (double*) malloc(n * sizeof(double));
26         env.tmp2 = (double*) malloc(n * sizeof(double));
27         return env;
28 }
29
30 int mean(double *x, double *z, double *m, int n, extrema_t *ex, envelope_t *env) {
31         int i;
32
33         extr(x, z, n, ex);
34
35         if (ex->n_min + ex->n_max < 7)
36                 return 1;
37
38         boundary_conditions(x, z, n, ex);
39
40         interpolation(env->e_max, ex->x_max, ex->y_max, ex->n_max, x, n);
41
42         interpolation(env->e_min, ex->x_min, ex->y_min, ex->n_min, x, n);
43
44         for (i = 0; i < n; i++)
45                 m[i] = (env->e_max[i] + env->e_min[i]) / 2;
46         return 0;
47 }
48
49 double emd_fabs(double x) {
50         if (x < 0)
51                 return -x;
52         else
53                 return x;
54 }
55
56 int mean_and_amplitude(double *x, double *z, double *m, double *a, short n,
57                 extrema_t *ex, envelope_t *env) {
58         int i;
59
60         extr(x, z, n, ex);
61
```

```
62          if (ex->n_min + ex->n_max < 3)
63                  return 1;
64
65          boundary_conditions(x, z, n, ex);
66
67          interpolation(env->e_max, ex->x_max, ex->y_max, ex->n_max, x, n);
68
69          interpolation(env->e_min, ex->x_min, ex->y_min, ex->n_min, x, n);
70
71          for (i = 0; i < n; i++) {
72                  m[i] = (env->e_max[i] + env->e_min[i]) / 2;
73          }
74
75          for (i = 0; i < n; i++) {
76                  a[i] = emd_fabs(env->e_max[i] - env->e_min[i]) / 2;
77          }
78
79          return 0;
80  }
```

```
1  /*
2   * operation.c
3   *
4   * Created on: Dec 19, 2011
5   * Author: Riccardo Del Re
6   *
7   * This code provides a set of useful functions
8   */
9  #include <stdio.h>
10 #define I(x,y,n) ((x)*(n))+(y)
11 /*
12  * return the absolut value of an element
13  */
14 double absol(double x) {
15         if (x < 0)
16                 return -x;
17         else
18                 return x;
19 }
20
21 /*
22  * find the range of an array (max(array)-min(array)
23  */
24 double findrange(double x[], int n){
25         int i;
26         double max,min;
27         max = x[0];
28         min = x[0];
29         for (i=1;i<n;i++){
30                 if (x[i]<min) min = x[i];
31                 if (x[i]>max) max = x[i];
32         }
33         return absol(max-min);
34 }
35 /*
36  * return the sum of a row array
37  */
38 double sum(double x[], int n){
39         int i;
40         double temp = 0;
41         for (i=0;i<n;i++) temp = temp + x[i];
42         return temp;
43 }
44
45 /*
46  * computes the matricial product between a matrix and a vector
47  */
48
49 double rcprod(double x[], double y[], int n){
50         double temp=0;
51         int i;
52         for (i=0;i<n;i++) temp = temp + x[i]*y[i];
53         return temp;
54 }
55
56 double mysqrt(double m)
57 {
58     double i=0;
59     unsigned int j;
60     double x1,x2;
61     while( (i*i) <= m )
```

```
62              i+=0.1;
63      x1=i;
64      for(j=0;j<10;j++)
65      {
66              x2=m;
67          x2/=x1;
68          x2+=x1;
69          x2/=2;
70          x1=x2;
71      }
72      return x2;
73  }
74
75  double findmax(double x[], int n){
76              int i;
77              double max;
78              max = x[0];
79              for (i=1;i<n;i++){
80                      if (x[i]>max) max = x[i];
81              }
82              return max;
83  }
84
85
86  double tolcalc(double sig[], double imf[], int numimfs, int n, double tol[],int start){
87              int i;
88              double TOL,range_sig,som_alpha;
89              double alpha[numimfs-start];
90              range_sig = findrange(sig,n);
91              TOL = 0.1*range_sig;
92              for (i=start;i<numimfs;i++){
93                      double temp[n];
94                      int j;
95                      for (j=0;j<n;j++) temp[j] = imf[I(i,j,n)];
96                      alpha[i-start] = range_sig/(findrange(temp,n));
97              }
98              som_alpha = sum(alpha,numimfs);
99              for (i=0;i<numimfs-start;i++) tol[i] = TOL*alpha[i]/som_alpha;
100             return TOL;
101 }
```

```
1  ///*
2  // * predict_ar.c
3  // *
4  // * Created on: Dec 21, 2011
5  // * Author: Riccardo Del Re
6  // */
7
8  void predict_ar(double ar_mcoefs[], short length_ar_mcoefs, short pred_hor,
9                  double xin[], double yp[]) {
10         short m_order = length_ar_mcoefs - 1;
11
12         short j;
13         for (j = 0; j < pred_hor; j++)
14                 yp[j] = 0.0;
15         for (j = 0; j < m_order; j++)
16                 yp[j] = xin[j];
17
18         short i;
19         for (i = m_order; i < pred_hor; i++) {
20
21                 yp[i] = 0.0;
22
23                 for (j = 0; j < m_order; j++) yp[i] = yp[i] - yp[i - j - 1] * ar_mcoefs[j + 1];
24     }
25 }
```

```
/*
 * ltc_encoder.c
 *
 * Created on: Feb 2, 2012
 * Author: rik
 */

#define I(x,y,n) (x)*(n)+(y)



int ltc_encoder(double x[], double tol, int N, double model[]){

    int index_ru=0;
    int index_rl=1;
    int index_lu=2;
    int index_ll=3;

    char flag=0;
    int n=0;

    double limit[N*4];

    int i=0;
    int j=0;

    double z=x[0];
    double ru=z;
    double rl=z;
    double lu=z;
    double ll=z;

    limit[I(i,index_ru,4)]=ru;
    limit[I(i,index_rl,4)]=rl;
    limit[I(i,index_lu,4)]=lu;
    limit[I(i,index_ll,4)]=ll;

    while (i<N){

        z = (lu+ll)/2;

        model[I(n,0,2)]=i;
        model[I(n,1,2)]=z;
        model(n,2)=z;
        n++;
        j=i+1;

        double r=x[j];

        ru = r+tol;
        rl = r-tol;
        lu = r+tol;
        ll = r-tol;

        limit[I(j,index_ru,4)]=ru;
        limit[I(j,index_rl,4)]=rl;
        limit[I(j,index_lu,4)]=lu;
        limit[I(j,index_ll,4)]=ll;

        flag=0;

```

```
62          while (flag==0 && j<N){
63
64              double lu_slope = (lu-z)/(j-i);
65              double ll_slope = (ll-z)/(j-i);
66
67              double lu_next = lu+lu_slope;
68              double ll_next = ll+ll_slope;
69
70              r = x[j+1];
71
72              double ru_next=r+tol;
73              double rl_next=r-tol;
74
75              if(lu_next<rl_next) flag=1;
76              if(ll_next>ru_next) flag=1;
77              if(flag==0){
78                  if(lu_next>ru_next){
79                      lu = ru_next;
80                  }
81                  else{
82                      lu = lu_next;
83                  }
84                  if(ll_next<rl_next){
85                      ll=rl_next;
86                  }
87                  else{
88                      ll=ll_next;
89                  }
90
91                  j=j+1;
92
93                  limit[I(j,index_ru,4)]=ru;
94                  limit[I(j,index_rl,4)]=rl;
95
96                  limit[I(j,index_lu,4)]=lu;
97                  limit[I(j,index_ll,4)]=ll;
98              }
99          }
100         i=j;
101     }
102         model[I(n,0,2)]=N;
103         model[I(n,1,2)]=(lu+ll)/2;
104         return n;
105 }
```

```
/*
 * ltc_ decoder.c
 *
 * Created on: Feb 2, 2012
 * Author: Riccardo Del Re
 */

#define I(x,y,n) (x)*(n)+(y)

void ltc_decoder(double model[],int n,int N,double y[]){
    int k=0;

    if (n==1){
        y[0] = model[I(0,1,2)];

    else{

        while(k<n){

            double x1 = model[I(k,0,2)];
            double x2 = model[I(k+1,0,2)];
            double y1 = model[I(k,1,2)];
            double y2 = model[I(k+1,1,2)];

            double m=(y2-y1)/(x2-x1);

            int i=(int)x1;
            while(i<=x2){
                y[i] = y1+m*(i-x1);
                i++;
            }
            k++;

        }
    }
}
```

```
/*
 * short_time_corr.c
 *
 * Created on: Feb 2, 2012
 * Author: Riccardo Del Re
 */

void short_time_corr(double x[], short x_dim, short time_lim, double ro[], double count[]){

    double m,sig;
        int i,j;
        m=0;
        for(i=0;i<x_dim;i++){
                m = m + x[i];
        }
        m = m/x_dim;

        sig = 0;
        for(i=0;i<x_dim;i++){
                sig = sig + (x[i]-m)*(x[i]-m);
        }
        sig = sig/x_dim;

        for(i=0;i<time_lim+1;i++) {
                count[i] = 0;
        }
        for(i=0;i<time_lim+1;i++){
                ro[i] = 0;
        }
        for(i=0;i<x_dim;i+=2){
                j=i;
                while(j<=i+time_lim && j<x_dim){

            ro[j-i] = ro[j-i] +(x[i]-m)*(x[j]-m)/sig;

                        count[j-i] = count[j-i]+1;
                        j++;
                }
        }
        for(i=0;i<time_lim+1;i++) ro[i] = ro[i]/count[i];
}
```

Bibliografia

[1] E. Huang et al., *The empirical mode decomposition and the Hilbert spectrum for nonlinear and non-stationary time series analysis*, 1996.

[2] G. Rilling, P. Flandrin, P. Gonçalvès, *On empirical mode decomposition and its algorithms.*

[3] N. Bui, M.Rossi, M. Zorzi, *Smart Grid Communication: Paradigm and Enabling Technologies*, 2011;

[4] G. Rilling, P. Flandrin, P. Gonçalvès, *EMD source code*, http://perso.ens-lyon.fr/patrick.flandrin/emd.html.

[5] A.Rosponi, M.Rossi, *Monitoraggio e controllo locale dei consumi elettrici in un smart microgrid*, Padova, 2011.

[6] T. von Eicken, D.E.Culler, S.C.Goldstein, K.E.Schauser, *Active messages: A mechanism for integrated communication and computation.*, Proceedings of the 19th Annual International Symposium on Computer Architecture, Queensland, Australia, 1992.

[7] T. Schoellhammer, B. Greenstein, E. Osterweil, M. Wimbrow, D. Estrin, *Lightweight temporal compression of microclimate datasets*, IEEE: International Conference on Local Computer Networks, 2004, pp 516-524.

[8] R.B.Davies, D.S.Harte, *Tests for Hurst effect*, Biometrica 74.

[9] D. Zordan, G. Quer, M.Zorzi, M.Rossi, *Modeling and Generation of Space-Time Correlated Signals for Sensor Network Fields*, IEEE Global Telecommunications Conference.

[10] N.Benvenuto, G. Cherubini, *Algorithms for Communications Systems and Their Applications*, Wiley, 2002.

[11] http://en.wikipedia.org/wiki/6LoWPAN.

[12] http://zolertia.sourceforge.net/wiki/index.php/Z1.

[13] http://www.ieee802.org/15/pub/TG4.html.

[14] http://en.wikipedia.org/wiki/IEEE_802.15.4.

[15] http://tools.ietf.org/html/rfc4944.

[16] B.Martinez, D.Zordan, I.Vilajosana, M.Rossi, *Temporal Data Compression for Wireless Sensor Networks based on Empirical Mode Decomposition* 2012.

[17] L.Bierl, *MSP430 Family Mixed-Signal Microcontroller Application Reports*, TEch. rep., Texas Instrument Incorporated, 2000.

[18] Chipcon, *SmartRF CC2420: 2.4GHz IEEE 802.15.4/ZigBee-ready RF Transceiver*, Tech. rep., Texas Instrument Incorporated (2007).

Ringraziamenti

Desidero ringraziare innanzitutto i miei genitori e la mia famiglia per avermi dato la possibilità di iniziare, continuare e terminare l'università, senza avermi mai messo pressione se qualcosa andava storto. Ringrazio il mio relatore prof. Michele Rossi per avermi dato la possibilità di svolgere questa attività, ed avermi sempre spronato a cercare metodi alternativi per risolvere i problemi e a perfezionare quello su cui lavoro. Ringrazio i ragazzi del laboratorio SIGNET del dipartimento, in particolare Riccardo Manfrin e Moreno Dissegna per la loro indiscussa capacità nel software, e Davide Zordan per avermi seguito e consigliato durante la parte di implementazione dell'algoritmo. Ringrazio inoltre Cristiano Tapparello, che mi ha dato un aiuto nella parte finale della tesi. Ringrazio poi tutti i miei amici che mi hanno supportato e sopportato durante questo ciclo di studi (in ordine casuale): Matteo Teo, Antonio, Serena, Davide, Maura, Sara, Matteo Cippi, Angelo, Laura, Nicola Rokkio, Alessandro, Paolo, Matteo M., Martina, Michele, Marco D., Marco C., Andrea, Cecilia, Giulia, Camilla, Eleonora, Chiara, Valentina, Fabio. Un ringraziamento speciale va a Paolo Conte e Marco Tomasin, senza di loro probabilmente non sarei ancora laureato.