Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea in Fisica

La trasformata veloce di Fourier (FFT): analisi e implementazione in C++

Relatore:
Prof. Fabio Ortolani

Presentata da:
Dario Lo Buglio

Sessione I
Anno Accademico 2014/2015

Abstract

La trasformata di Fourier (FT) è uno strumento molto potente implementato, oggi, in un enorme numero di tecnologie. Il suo primo esempio di applicazione fu proprio il campionamento e la digitalizzazione di segnali analogici. Nel tempo l'utilizzo della FT è stato ampliato a più orizzonti in ambito digitale, basti pensare che il formato di compressione '.jpg' utilizza una FT bidimensionale, mentre uno degli ultimi esempi di applicazione si ha nell'*imaging digitale* in ambito medico (risonanza magnetica nucleare, tomografia assiale computerizzata TAC ecc...). Nonostante gli utilizzi della FT siano molto diversificati il suo basilare funzionamento non è mai cambiato: essa non fa altro che modificare il dominio di una funzione del tempo (un segnale) in un dominio delle frequenze, permettendo così lo studio della composizione in termini di frequenza, ampiezza e fase del segnale stesso. Parallelamente all'evoluzione in termini di applicazioni si è sviluppato uno studio volto a migliorare e ottimizzare la computazione della stessa, data l'esponenziale crescita del suo utilizzo. In questa trattazione si vuole analizzare uno degli algoritmi di ottimizzazione più celebri e utilizzati in tal senso: la trasformata veloce di Fourier (Fast Fourier Transformation o FFT). Si delineeranno quindi le caratteristiche salienti della FT, e verrà introdotto l'algoritmo di computazione tramite linguaggio C++ dedicando particolare attenzione ai limiti di applicazione di tale algoritmo e a come poter modificare opportunamente la nostra funzione in modo da ricondurci entro i limiti di validità.

Indice

Capitolo 1

La trasformata di Fourier

1.1 Introduzione e caratteri generali

La **trasformata di Fourier (FT)** prima di essere un utile strumento di analisi spettrale è una trasformazione matematica che ad una funzione f

$$f : \Re^n \to C$$

formalmente fa corrispondere una F

$$F(\xi) = (Ff)(\xi)) = \frac{1}{(2\pi)^{\frac{n}{2}}} \int_{\Re^n} e^{-i\xi x} f(x)dx \tag{1.1}$$

Si definisce automaticamente anche la **antitrasformata di Fourier**

$$F(-\xi) = (\widetilde{F}f)(\xi)) = \frac{1}{(2\pi)^{\frac{n}{2}}} \int_{\Re^n} e^{i\xi x} f(x)dx \tag{1.2}$$

Con ξ e $x \in \Re^n$. Un segnale è una funzione h che assume valori dipendenti dal tempo *h(t)*; alternativamente, se utilizziamo il dominio delle frequenze, si può pensare un segnale come una ampiezza H (generalmente complessa e che include anche una fase iniziale) dipendente dalla frequenza ν e dunque *H(ν)*. La trasformata di Fourier di un segnale dipendente dal tempo restituisce infatti la distribuzione in frequenze del segnale stesso. Quindi per una funzione $h(t)$ vale

$$H(\nu) = \int_{-\infty}^{+\infty} h(t)e^{2\pi i\nu t}dt \tag{1.3}$$

$$h(t) = \int_{-\infty}^{+\infty} H(\nu)e^{-2\pi i\nu t}dt \tag{1.4}$$

Se ne deduce che la FT gode della proprietà di linearità (la FT di una somma di funzioni è la somma delle singole FT delle due funzioni) e che il prodotto per una

costante di una funzione di cui si vuole calcolare la FT è il prodotto della costante per la FT della funzione. Inoltre se la $h(t)$ gode di proprietà quali parità, disparità, queste si riflettono completamente sulla FT. La tabella seguente esemplifica i più comuni casi.

Proprietà	**Proprietà della FT**
$h(t)$ è reale	$H(\nu) = H(-\nu)$ è pari
$h(t)$ è immaginaria	$H(\nu) = -H(\nu)$ è dispari
$h(t)$ è pari	$H(\nu) = H(-\nu)$ è pari
$h(t)$ è dispari	$H(\nu) = -H(\nu)$ è dispari
$h(t)$ è reale e pari	$H(\nu)$ è reale e pari
$h(t)$ è reale e dispari	$H(\nu)$ è immaginaria e dispari
$h(t)$ è immaginaria e pari	$H(\nu)$ è immaginaria e pari
$h(t)$ è immaginaria e dispari	$H(\nu)$ è reale e dispari

Ci sono poi le proprietà di shifting e rescaling

$$h(at) \iff \frac{1}{|a|}H(\frac{\nu}{a}) \quad Time \quad scaling$$

$$\frac{1}{|b|}h(\frac{t}{b}) \iff H(\nu b) \quad Frequency \quad scaling$$

$$h(t - t_0) \iff H(\nu)e^{2\pi i\nu t_0} \quad Time \quad shifting$$

$$h(t)e^{-2\pi i\nu_0 t} \iff H(\nu - \nu_0) \quad Frequency \quad shifting$$

Quando si analizza un segnale tramite la FT, un attento studio delle proprietà del segnale (tramite le formule appena elencate) permette una più rapida ed efficiente digitalizzazione nonché elaborazione del segnale. Graficamente, quello che succede è esemplificato visualmente

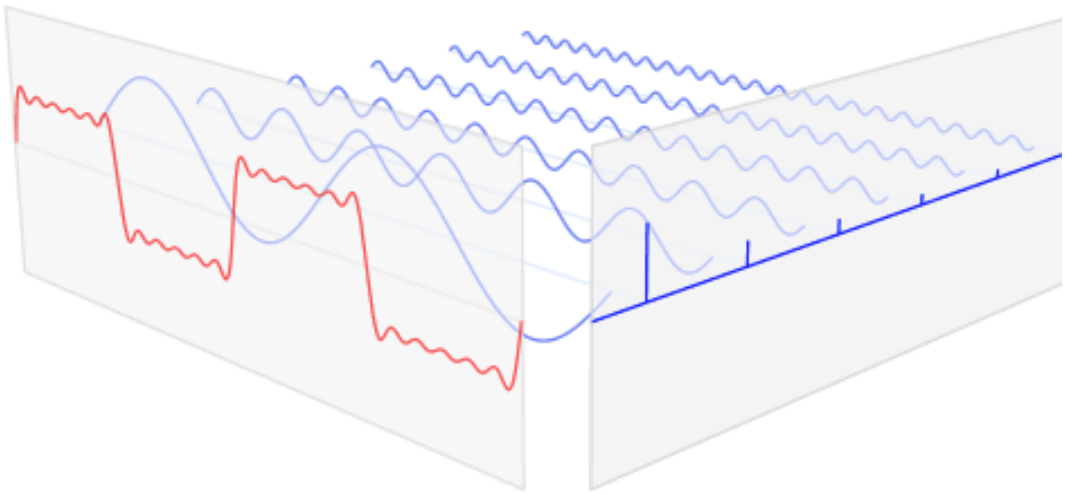

Figura 1.1: Dominio del tempo e della frequenza

Dove sulla sinistra abbiamo, nel dominio del tempo, una funzione d'onda quadra espressa come somma delle prime sei armoniche della serie di Fourier, e sulla destra, nel dominio delle frequenze, la composizione del segnale. Con una grossa mole di dati e segnali non così semplici come l'onda quadra si ottengono invece grafici molto più complicati.

1.1.1 Teoremi di convoluzione, correlazione e Parseval

Con la definizione data di FT possiamo enunciare dei teoremi molto importanti che riguardano operazioni tra funzioni. Prese $g(t)$ e $h(t)$ si definisce la **convoluzione tra due funzioni**:

$$g \star h = \int_{-\infty}^{\infty} g(\tau)h(t-\tau)d\tau \tag{1.5}$$

rappresentante il valore dell'area sottesa dal prodotto delle due funzioni nel tempo man mano che una venga traslata sopra l'altra. Si definisce inoltre la **correlazione** tra due funzioni:

$$Corr(g,h) = \int_{-\infty}^{\infty} g(\tau+t)h(\tau)d\tau \tag{1.6}$$

Le due operazioni sono molto simili tra di loro e si equivalgono se le funzioni sono simmetriche rispetto ai valori di traslazione cui sono sottoposte. Un risultato notevole per questi due tipi di operazioni è che la FT della convoluzione tra due funzioni è il prodotto delle FT delle singole funzioni, e dunque la FT della correlazione tra due funzioni è il prodotto della FT di una per la FT complesso-coniugata dell'altra.

In simboli

$$g \star h \Leftrightarrow G(\nu)H(\nu) \qquad\qquad Corr(g,h) \Leftrightarrow G(\nu)H^*(\nu)$$

Questi risultati sono molto comodi vista la semplicità del calcolo una volta passati nel dominio delle frequenze. Un ultimo risultato importante invece è dato dal **Teorema di Parseval**:

Teorema 1.1 *La potenza totale di un segnale è indipendente dal dominio in cui la calcolo, sia esso quello del tempo o quello delle frequenze, dunque:*

$$P_{tot} = \int_{-\infty}^{\infty} |h(t)|^2 dt = \int_{-\infty}^{\infty} |H(\nu)|^2 d\nu.$$

Capitolo 2

Sampling dei dati e discretizzazione della FT

2.1 Acquisizione del segnale

Il primo passo nel processo di conversione da un segnale analogico a un segnale digitale è il campionamento dei dati, cioè l'estrapolazione di N valori (samples) del segnale ad intervalli di tempo regolari. Chiameremo allora *tempo di campionamento* Δ l'intervallo di tempo tra la registrazione di un valore e il successivo, e *sampling rate* il suo reciproco (frequenza di campionamento) $\nu_c = 1/\Delta$. Di conseguenza, preso un segnale dipendente dal tempo $h(t)$, possiamo scrivere i samples come

$$h_n = h(n\Delta)$$

con n intero.
Uno dei risultati notevoli nella digitalizzazione è il **teorema di Nyquist-Shannon**:

Teorema 2.1 *Se una funzione continua nel tempo h(t), campionata ad un determinato intervallo Δ , è limitata in frequenza dalla ν_{max}, allora il segnale è univocamente determinato dall'equazione*

$$h(t) = \sum_{n=-\infty}^{+\infty} h_n \frac{sin[\pi\nu_c(t - n\Delta)]}{\pi(t - n\Delta)} \tag{2.1}$$

se la frequenza di campionamento $\nu_c \geq 2\nu_{max}$. Chiamiamo allora ***frequenza critica di Nyquist*** *la metà della frequenza di campionamento $\nu_n = \frac{\nu_c}{2} = \frac{1}{2\Delta}$*

Questo importante teorema ci suggerisce un modo molto utile e pratico per ricostruire un segnale analogico partendo dai suoi samples ma ci mette in allarme riguardo il problema noto dell'aliasing. In particolare se la ν_{max} è superiore alla ν_n

succede che i valori di $h(t)$ corrispondenti all'esterno dell'intervallo $[-\nu_n, +\nu_n]$ vengono traslati all'interno di tale range provocando una distorsione del segnale più o meno notevole. Ad esempio immaginiamo di avere una funzione qualunque e costruiamone la FT ipotetica:

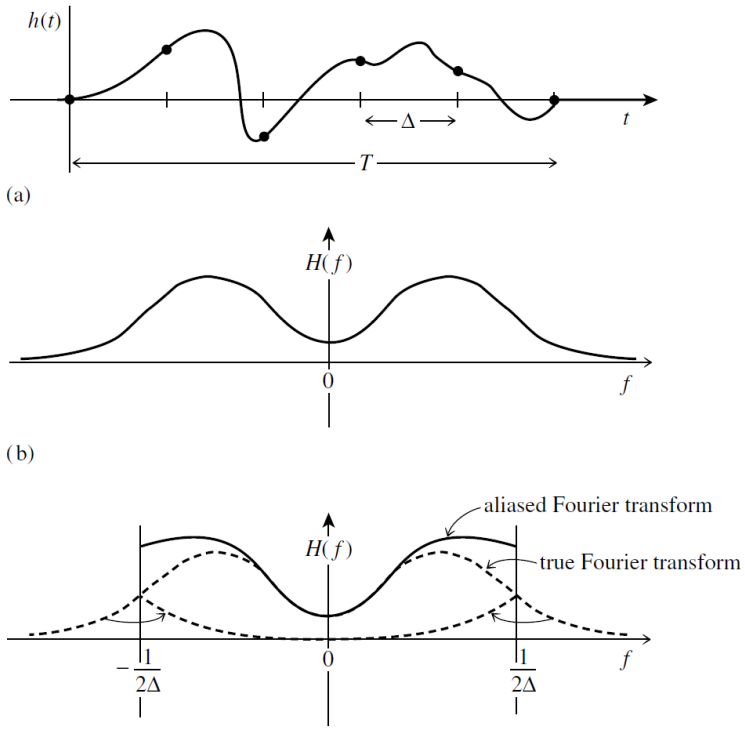

Figura 2.1: Aliasing

Si può vedere come i segnali esterni all'intervallo della frequenza di Nyquist vengano sommati internamente provocando una distorsione notevole. Per risolvere il problema dell'aliasing le soluzioni sono diverse: o si usa un cosiddetto filtro passa basso (passivo come RC, o attivo) oppure si utilizzano strumenti di campionamento con frequenze maggiori. Se non si conosce la reale limitatezza in frequenza della funzione, l'unico metodo che abbiamo per sapere se la nostra FT è affetta da aliasing oppure no è guardare il comportamento della trasformata stessa quando questa si avvicina ai limiti dell'intervallo di frequenza di Nyquist: se la FT si avvicina a 0 andando verso i limiti allora è molto probabile che abbiamo minimizzato l'effetto di aliasing, se invece stimiamo che la FT si avvicina ad un valore costante diverso da 0 allora è probabile che siamo in presenza di un effetto di aliasing e quindi la nostra FT è parzialmente distorta.

2.2 Discretizzazione della FT

Una volta analizzato il metodo di acquisizione bisogna fare una trattazione sul come discretizzare una trasformata matematica di per sé definita continua come quella di Fourier. Supponiamo adesso di avere campionato il segnale in ingresso e avere ottenuto gli N samples consecutivi (per semplicità assumiamo N pari)

$$h_k = h(t_k) \qquad con \qquad t_k = k\Delta \qquad e \qquad k = 0, 1, 2...N-1$$

Se la funzione è diversa da zero solo in un intervallo finito di tempo, allora supponiamo che l'intero intervallo temporale ricoperto da i nostri samples contenga tutti i valori della funzione diversi da zero. Se invece la funzione è periodica, o è sempre diversa da zero, supponiamo che i nostri samples contengano un numero sufficiente di periodi (per funzioni periodiche) o che quantomeno diano una rappresentanzione sufficiente dell'andamento tipico della funzione. Ovviamente con N samples di una funzione continua non possiamo fare altro che calcolare N valori di uscita, quindi anziché concentrarci sulla funzione continua concentriamoci sui suoi N valori; questo vuol dire che, considerando la FT di $h(t)$ ovvero $H(\nu)$, non consideriamo tutti i suoi valori continui ma soltanto i valori discreti di frequenze:

$$\nu_n = \frac{n}{N\Delta} \qquad con \qquad n = -\frac{N}{2}, -\frac{N}{3}... + \frac{N}{2}$$

Verrebbe da pensare che così il conto dei samples in realtà sia N+1 a giudicare da n ma in realtà i limiti superiori e inferiori di n non sono indipendenti come gli altri poiché corrispondono alla frequenza di Nyquist. Dunque se consideriamo solo questo set discreto di frequenze allora possiamo approssimare la FT nel seguente modo:

$$H(\nu_n) = \int_{-\infty}^{\infty} h(t) e^{2\pi i \nu_n t} dt \simeq \sum_{k=0}^{N-1} h_k e^{2\pi i \nu_n t_k} \Delta = \Delta \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \tag{2.2}$$

l'ultima uguaglianza definisce la **Trasformata discreta di Fourier (DFT)** degli N punti h_k che possiamo scrivere dunque come:

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \tag{2.3}$$

e dunque

$$H(\nu_n) \simeq \Delta H_n$$

Cioè quello che succede è che la DFT mappa gli N numeri complessi h_k negli N numeri complessi H_n . Continuano a valere le proprietà viste nella tabella successiva

alle equazioni 1.3 e 1.4 e possiamo facilmente ricavare anche la **antitrasformata discreta di Fourier** con poche modifiche alla equazione 2.3 (comodo dal punto di vista computazionale):

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i k n/N} \tag{2.4}$$

e valgono ancora (in forma discreta) i teoremi di convoluzione, correlazione e Parseval. A titolo d'esempio il teorema di Parseval in forma discreta diventa:

$$\sum_{k=0}^{N-1} |h_k|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |H_n|^2 \tag{2.5}$$

Capitolo 3

Fast Fourier Transform (FFT)

3.1 Algoritmo di fattorizzazione di Cooley-Tukey

Per trovare un algoritmo che minimizzi il numero di operazioni da effettuare per ottenere la DFT bisogna prima vedere, nel suo caso più semplice, quante e quali siano tali operazioni. Una volta campionati gli N samples e definito un numero complesso

$$W_N = e^{2\pi i/N}$$

possiamo riscrivere l'equazione 2.3 come

$$H_n = \sum_{k=0}^{N-1} h_k W^{kn}$$

Ovvero, il vettore h_k viene moltiplicato per una matrice NxN W^{nk} in cui n è fissato da H_n e k si estende alla somma. Quindi le operazioni totali da svolgere per ottenere l'intero vettore H_n sono N^2 moltiplicazioni tra numeri complessi più un limitato numero di operazioni per calcolare effettivamente i vari elementi della matrice (potenze di numeri complessi). Storicamente, i primi che trovarono algoritmi semplificativi, si accorsero che N^2 poteva essere enormemente ridotto fino a $Nlog_2(N)$. Per avere un'idea dell'abissale differenza tra i due calcoli si pensi ad $N = 10^8$ che porta rispettivamente a 10^{16} operazioni calcolate come N^2 contro $10^8 * log_2(10^8) \simeq 10^8 * 26,57$. Si capisce dunque come l'ottimizzazione di tale algoritmo sia fondamentale per un velocissimo calcolo. Esistono varie versioni e livelli di ottimizzazione del calcolo ma l'idea più chiara e semplice è sicuramente quella basata sul **lemma di Danielson-Lanczos**: *"Una trasformata di Fourier discreta di lunghezza N può essere riscritta come la somma di due trasformate di Fourier discrete di lunghezza N/2, una formata dai termini di indice pari e una formata dai termini di indice dispari "* :

$$H_n = \sum_{j=0}^{N-1} e^{\frac{2\pi ijn}{N}} h_j = \sum_{j=0}^{N/2-1} e^{\frac{2\pi i(2j)n}{N}} h_{2j} + \sum_{j=0}^{N/2-1} e^{\frac{2\pi i(2j+1)n}{N}} h_{2j+1} = \tag{3.1}$$

$$= \sum_{j=0}^{N/2-1} e^{\frac{2\pi i 2jn}{N/2}} h_{2j} + W_N^n \sum_{j=0}^{N/2-1} e^{\frac{2\pi ijn}{N/2}} h_{2j+1} = F_n^p + W_N^n F_n^d$$

Dove con p si indicano i termini "pari" e con d i termini "dispari". Di per sé il risultato appena ottenuto non lascia emergere la sua reale potenza che invece si fa notare subito se consideriamo che possiamo iterare il procedimento appena effettuato per i primi due termini. Notiamo che i due termini ottenuti sono DFT di lunghezza $N/2$; se iteriamo il processo possiamo arrivare ad ottenere un numero N di termini che rappresentano delle DFT di lunghezza unitaria. A questo punto bisogna capire cosa è una DFT di lunghezza unitaria: semplicemente il copiare l'input in output. Quello che otteniamo dunque è una combinazione lineare degli input h_j pesati dai numeri complessi W_N. Vediamo di fare un esempio esplicativo con l'intento però di far trasparire il procedimento generale per N molto grandi.

3.1.1 Un esempio con N=8

Ricordando che $W_N = e^{2\pi i/N}$

Prendiamo $N = 8$ e, riferendoci alla 3.1, scriviamo:

$$H_n = \sum_{j=0}^{8-1} e^{\frac{2\pi ijn}{8}} h_j = \sum_{j=0}^{4-1} e^{\frac{2\pi i(2j)n}{8}} h_{2j} + \sum_{j=0}^{4-1} e^{\frac{2\pi i(2j+1)n}{8}} h_{2j+1} =$$

$$= \sum_{j=0}^{4-1} e^{\frac{2\pi ijn}{4}} h_{2j} + W_8^n \sum_{j=0}^{4-1} e^{\frac{2\pi ijn}{4}} h_{2j+1}$$

divido i due termini ulteriormente in termini pari-pari pp, pd, dp e dd

$$H_n = \sum_{j=0}^{2-1} e^{\frac{2\pi i(2j)n}{4}} h_{4j} + \sum_{j=0}^{2-1} e^{\frac{2\pi i(2j+1)n}{4}} h_{4j+2} + W_8^n \sum_{j=0}^{2-1} e^{\frac{2\pi i(2j)n}{4}} h_{4j+1} +$$

$$+ W_8^n \sum_{j=0}^{2-1} e^{\frac{2\pi i(2j+1)n}{4}} h_{4j+3} = \sum_{j=0}^{2-1} e^{\frac{2\pi ijn}{2}} h_{4j} + W_4^n \sum_{j=0}^{2-1} e^{\frac{2\pi ijn}{2}} h_{4j+2} +$$

$$+ W_8^n \sum_{j=0}^{2-1} e^{\frac{2\pi ijn}{2}} h_{4j+1} + W_8^n W_4^n \sum_{j=0}^{2-1} e^{\frac{2\pi ijn}{2}} h_{4j+3}$$

e facciamo un'ultima iterazione per arrivare ai termini finali ppp, ppd, pdp, pdd, dpp, dpd, ddp e ddp.

$$H_n = \sum_{j=0}^{1-1} e^{\frac{2\pi i(2j)n}{2}} h_{8j} + \sum_{j=0}^{1-1} e^{\frac{2\pi i(2j+1)n}{2}} h_{8j+4} + W_4^n \sum_{j=0}^{1-1} e^{\frac{2\pi i(2j)n}{2}} h_{8j+2} +$$

$$+W_4^n \sum_{j=0}^{1-1} e^{\frac{2\pi i(2j+1)n}{2}} h_{8j+6} + W_8^n \sum_{j=0}^{1-1} e^{\frac{2\pi i(2j)n}{2}} h_{8j+1} + W_8^n \sum_{j=0}^{1-1} e^{\frac{2\pi i(2j+1)n}{2}} h_{8j+5} +$$

$$+W_8^n W_4^n \sum_{j=0}^{1-1} e^{\frac{2\pi i(2j)n}{2}} h_{8j+3} + W_8^n W_4^n \sum_{j=0}^{1-1} e^{\frac{2\pi i(2j+1)n}{2}} h_{8j+7} =$$

$$= h_0 + W_2^n h_4 + W_4^n h_2 + W_4^n W_2^n h_6 + W_8^n h_1 + W_8^n W_2^n h_5 + W_8^n W_4^n h_3 + W_8^n W_4^n W_2^n h_7$$

Notiamo che, così facendo, abbiamo implicitamente inserito un algoritmo di *reverse bit*. Infatti associando lo stato logico "0" ad un termine "pari" e lo stato logico "1" ad un termine "dispari" abbiamo che, in ordine per come sono stati trovati, gli h_n sono:

Termine numero	**che in binario è**	**combinazione di pari e dispari**
0	000	000 (ppp)
1	001	100 (dpp)
2	010	010 (pdp)
3	011	110 (ddp)
4	100	001 (ppd)
5	101	101 (dpd)
6	110	011 (pdd)
7	111	111 (ddd)

e il *reverse bit* si manifesta nel fatto che le combinazioni della seconda e terza colonna sono le medesime combinazioni di bit ma invertite specularmente.
Analizziamo adesso i coefficienti W_N^n; questi si capisce bene come siano periodici in n. Inoltre è possibile portarli tutti in forma unica scrivendoli come W_8^{jn} sfruttando la proprietà

$$W_{8/j}^n = W_8^{jn}$$

e quindi riferendoci agli otto precedenti valori calcolati osserviamone i valori per n=1 a titolo d'esempio:

1	W_8^0	1
$W_2^n = W_{8/4}^n$	W_8^{4n}	-1
$W_4^n = W_{8/2}^n$	W_8^{2n}	i
$W_4^n W_2^n = W_{8/2}^n W_{8/4}^n$	W_8^{6n}	$-i$
$W_8^n = W_8^n$	W_8^n	$e^{i\pi/4}$
$W_8^n W_2^n = W_8^n W_{8/4}^n$	W_8^{5n}	$-e^{i\pi/4}$
$W_8^n W_4^n = W_8^n W_{8/2}^n$	W_8^{3n}	$e^{i3\pi/4}$
$W_8^n W_4^n W_2^n = W_8^n W_{8/2}^n W_{8/4}^n$	W_8^{7n}	$-e^{i3\pi/4}$

Il calcolo di questi coefficienti richiede in genere un numero di operazioni trascurabili rispetto al calcolo principale dei singoli termini della FFT, che abbiamo detto essere $Nlog_2N$. Prima di vedere però il perché di tale risultato introduciano una rappresentazione grafica tramite i cosidetti *butterfly diagrams* descrivendone prima il loro simbolismo:

Figura 3.1: Butterfly diagram

La freccia verso l'alto indica una somma e la freccia verso il basso indica una sottrazione; il fattore W_N^j indica che prima di fare l'operazione di somma o sottrazione la variabile b viene moltiplicata per tale valore. Quindi gli output saranno la somma (in alto) e la differenza (in basso) tra a e $W_N^j b$. Vale la pena allora verificare che l'algoritmo di FFT secondo Cooley-Tukey può essere rappresentato complessivamente dal seguente *butterfly diagram* per $N = 8$

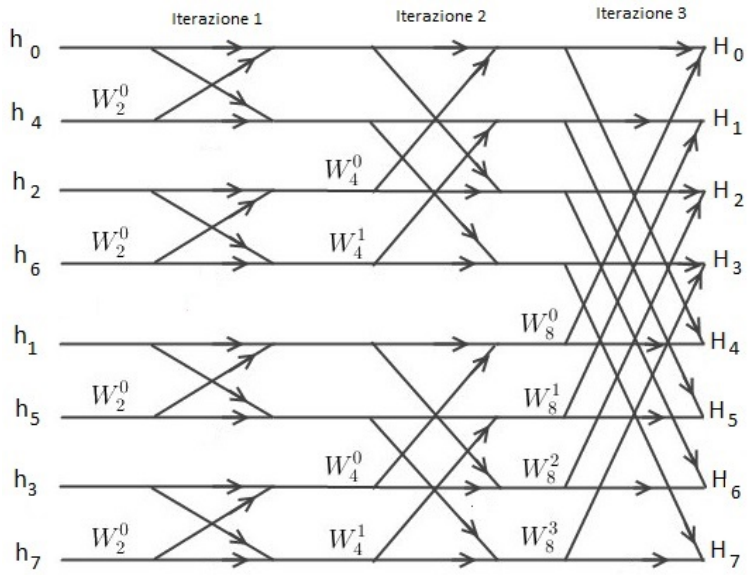

Figura 3.2: Butterfly diagram per N=8

Questo risultato visuale ci aiuta a capire infine il numero effettivo di calcoli da effettuare. Notiamo che abbiamo 3 iterazioni ognuna composta da 8 somme; non ci stupirà quindi constatare che il numero di calcoli da effettuare effettivamente è $Nlog_2(N) = 8log_2(8) = 24$ più un numero di operazioni (in genere irrilevante per N grandi) per calcolare i vari W_N^n. Una attenta lettura di quanto detto precedentemente ci fa intuire come questo algoritmo valga soltanto per un numero di samples che sia potenza di 2. Effettivamente con un numero che non è potenza di 2 non è possibile trovare un numero di iterazioni pari a log_2N e quindi non varrebbe più il metodo sviluppato. Una soluzione pratica nel caso di un numero di samples che non è potenza di 2, è quella di riempire il nostro pattern di dati con tanti termini nulli fino alla potenza di 2 successiva. Questa soluzione non introduce nessuna perturbazione al segnale perchè ovviamente i termini nulli inseriti non contribuiscono in nessun modo al risultato finale della FFT. E' questo d'altronde l'unico limite di applicazione dell'algoritmo. Esistono algoritmi diversi, che esulano dai nostri scopi, per ogni numero di samples possibile, anche per numeri primi. Ovviamente per $N = 8$ è stato possibile scrivere ricorsivamente la soluzione, ma capiamo subito che per N molto grandi è necessario trovare una soluzione computazionale in un linguaggio di programmazione che ci permetta di fare, molto velocemente, il calcolo di una FT tramite l'algoritmo della FFT, dato il suo enorme risultato nel diminuire il numero di operazioni da svolgere effettivamente.

3.2 Implementazione in C++

Il C++ è uno dei linguaggi di programmazione che più si adatta ai nostri scopi vista la sua grande versatilità nell' *Object oriented* e quindi nella riutilizzabilità del codice. Scelto il linguaggio di programmazione, costruiamo un programma che chieda all'utente di inserire i samples manualmente, ma si capisce bene come basti sostituire l'inserimento manuale con una funzione che raccolga automaticamente i dati da una sorgente esterna o da un file esterno con le classiche funzioni di lettura e scrittura dello standard.

```
#include<iostream>
#include<complex>
#include<fstream>
#include<sstream>
#include<vector>
#include<stdlib.h>
#include<math.h>
#define PI 3.14159265358979323846
using namespace std;

int log2(int N)  //logaritmo in base 2 di un intero
{
   int k = N, i = 0;
   while(k) {k >>= 1; i++;}
   return i-1;
}

int potenza2(int n)//la funzione controlla se il numero in argomento è una potenza
{                  //di due e in caso non lo sia genera h che è la potenza di due
   int h=n;        //successiva a n dato. infine la funzione assume come valore o n
                   //o la potenza piu' vicina h
   if(!(n > 0 && (n & (n - 1)))){ return n;}
   else
   {
      while (h & (h-1))
      {
         h = h & (h-1);
      }
      h = h << 1;
   }
   return h;
};

int invertibit(int N, int n)//N è numero di samples e n è l'indice intero
{                           //(in bit) che rappresenta i singoli N
   int j, r = 0;            //la funzione assume come valore il reverse bit di n
   for(j = 1; j <= log2(N); j++)
   {
      if(n & (1 << (log2(N) - j)))
      {r |= 1 << (j - 1);}
   }
   return r;
};
```

```
48
49  void ordina(complex<double>* vec, int n)//dispone gli elementi del vettore
50                                          //ordinandoli in ordine dato dal reverse
51  {                                       //bit precedente
52     const int c=n;
53     complex<double> vec2[c];
54     for(int i = 0; i < n; i++)
55     {vec2[i] = vec[invertibit(n, i)];}
56     for(int j = 0; j < n; j++)
57     {vec[j] = vec2[j];}
58  }
59
60  void FFT(complex<double>* vec1, int N, double t)
61  {
62     complex<double> W[N / 2];        //vettore dei pesi NB sono sempre in numero N/2
63                                      //(di cui uno è sempre 1).
64     W[1] = polar(1., 2. * M_PI / N); //polar restituisce (1*cost(theta),1*sin(theta)
65     W[0] = 1;                        //ovvero la forma cartesiana
66     for(int i = 2; i < N / 2; i++)
67     {
68        W[i] = pow(W[1], i);    //calcolo i vari W e li assegno al vettore dei pesi
69     }
70     int n = 1;
71     int a = N / 2;
72     for(int j = 0; j < log2(N); j++)//Comincia l'iterazione del lemma di
73                                     //Danielson-Lanczos
74     {                               //log2N iterazioni
75        for(int i = 0; i < N; i++)   //ognuna per N volte ---> Nlog2(N)
76        {
77           if(!(i & n))
78           {
79              complex<double> temp = vec1[i];
80              complex<double> Temp = W[(i * a) % (n * a)] * vec1[i + n];
81              vec1[i] = temp + Temp;      //NB il ripetersi con periodo N/2 dei pesi
82                                          //ma con segno opposto fa si che ci
83              vec1[i + n] = temp - Temp;  //siano N/2 addizioni e N/2 sottrazioni
84           }
85        }
86        n *= 2;    //Ad ogni passo n aumenta.. ad esempio con N=8 n=1,2,4
87        a = a / 2; //e lo stesso per a.. con N=8 a=4,2,1
88     }
89     for(int k=0; k<N; i++) {vec1[k]*=(t*sqrt(N));}
90                                         //moltiplico il vettore trasformato (DFT)
91  }                                      //per il tempo di campionamento e ottengo
92                                         //la FFT definitiva
93  int main ()
94  {
95     double t; //tempo di campionamento
96     int s; //numero samples iniziali
97     cout << "Inserire il tempo di campionamento (s): " << endl;
98     cin >> t;
99     cout << "Inserire il numero di samples: " << endl; cin >> s;
100    int r=potenza2(s); //r=s se s è potenza di 2 altrimenti la potenza di due
101                       //successiva
102    int d=r-s;         //d è 0 se r=s altrimenti è il numero di 0 da aggiungere al
103                       //set di dati
104    const int c=r;
105    complex<double> vec1[c];
106    cout << "Inserire i dati nella forma 'reale immaginaria': " << endl;
107
108
109
```

```
110
111
112     for(int i = 0; i < s; i++) //RIEMPIO IL VETTORE CON I DATI
113     {
114        cout << "inserisci la componente " << i+1 << endl;
115        double real, imag;
116        cin >> real >> imag;
117        complex<double> p(real, imag);
118        vec1[i]=p;
119        p=0; real=0; imag=0;
120     }
121     for(int j=s; j<r-1; j++) //RIEMPIO DI 0 i rimanenti termini
122     {
123        vec1[j]=0;
124     }
125     //for(int j=0; j<r; j++) {cout << vec1[j] << " ";} //vettore non invertito
126     //cout << " " << endl;
127     ordina(vec1,r);
128     //for(int j=0; j<r; j++) {cout << vec1[j] << " ";} //vettore invertito
129     //cout << " " << endl;
130     FFT(vec1,r,t);
131     //for(int j=0; j<r; j++) {cout << vec1[j] << " ";} //vettore trasformato
132     //cout << " " << endl;
133     system("pause");
134     return 0;
135 }
```

Per testare il funzionamento del software è stato rimpiazzato l'inserimento manuale dei dati con una generazione automatica delle funzioni di cui abbiamo poi graficato i risultati. A titolo d'esempio si riporta la funzione *main* nel caso di una funzione sinusoidale di frequenza *3 Hz* :

```
1  int main ()
2  {
3  int s=2048; //numero samples iniziali per semplicita' fissato ad un valore
4  const int c=s;
5  double j=0;
6  complex<double> vec1[c];
7  for(int i=0; i<s; i++)
8  {
9  vec1[i]=sin(6*PI*j); //Frequenza 3Hz, j è definita sopra
10 j+=0.00048828125;    // 1/2048 .. frequenza di campionamento di 2kHz
11 }
12 ordina(vec1,s);
13 double t=0.00048828125; //tempo di campionamento per la costante
14 FFT(vec1,s,t);      //di normalizzazione della DFT
15 ofstream output; //Scrivo in output i valori trasformati
16 output.open("C:\\INDIRIZZO_DI_SALVATAGGIO_\\file.csv");
17 for(int i=0; i<((s/2)); i++)  //stabilisco l'ordine di scrittura
18 {
19 output << abs(vec1[s/2+i]) <<endl; //modulo del complesso
20 }
21 for(int i=0; i<(s)/2; i++)
22 {
23 output << abs(vec1[i]) <<endl; //prima scrivo la seconda meta'
24 }                               //e infine la prima meta'
25 output.close();
26 system("pause");
27 return 0;}
```

Abbiamo utilizzato e graficato il $sin(2\pi 3t)$ e $sin(2\pi 8t)$ utilizzando una frequenza di campionamento di $2048Hz$ prendendo soltanto 1 secondo di campionamento (quindi $N = 2048$). Abbiamo deciso di utilizzare il modulo del complesso in uscita per semplicità, ma bisogna considerare che, a volte, si preferisce utilizzare il modulo quadro del complesso per avere dei risultati in termini energetici (basti pensare al teorema di Parseval enunciato sopra). I risultati sono mostrati dai seguenti grafici (per ovvie ragioni la scala sulle ascisse è stata allargata soltanto alla zona d'interesse):

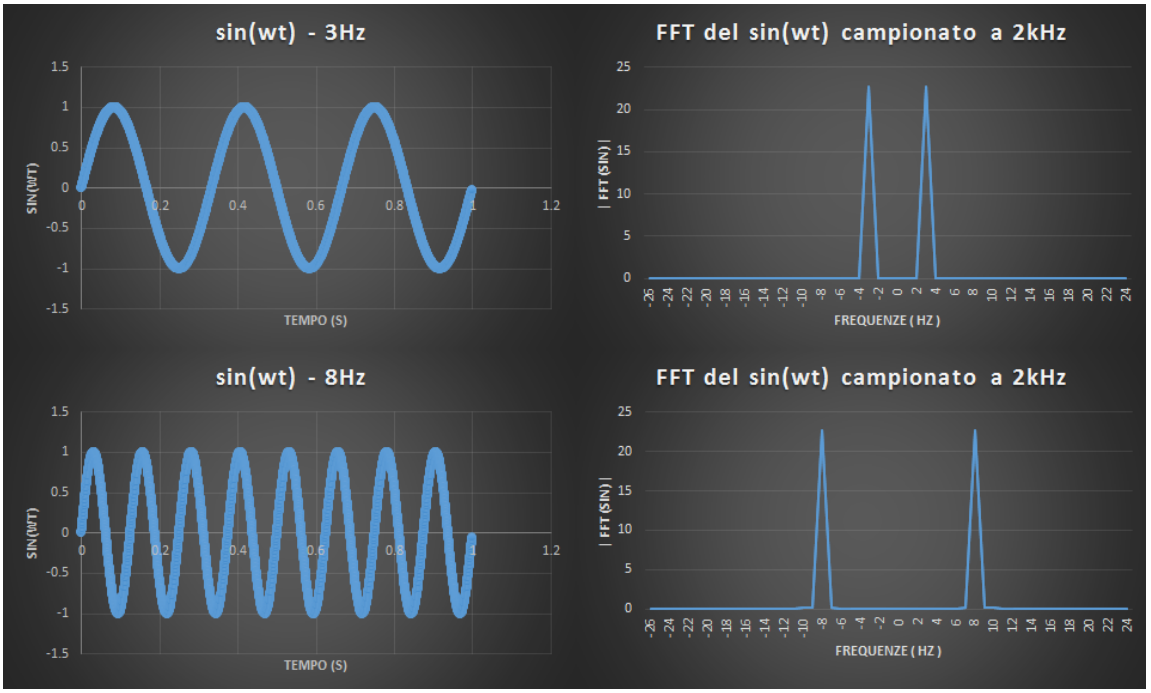

Figura 3.3: Due sinusoidi e loro trasformate

Come ci si aspettava abbiamo ritrovato le frequenze utilizzate nella generazione delle funzioni, a dimostrazione del corretto funzionamento del calcolo. L'avere dei picchi così pronunciati e dei valori che vanno repentinamente a zero è dovuto al fatto che abbiamo campionato dei segnali di pochi Hz con una frequenza di tre ordini di grandezza superiore; questo riduce sensibilmente il rumore e assicura una altissima precisione nella collezione dei dati. Inoltre è da tenere in considerazione il fatto che i valori in uscita sulle ordinate sono così alti grazie anche ad una normalizzazione della DFT tramite la costante di tempo di campionamento (come da eq. 2.3) e la radice del numero di samples (come da eq. 2.5). Ovviamente possiamo avere una funzione composta esprimibile come serie di seni e coseni; allora in questo caso basterà sommare i singoli termini e calcolarne un unica FFT; quello che ci si aspetta è di ritrovare tutte le frequenze che compongono il segnale (scopo principale della trasformata di Fourier appunto). Ad esempio sommando le due precedenti funzioni otteniamo:

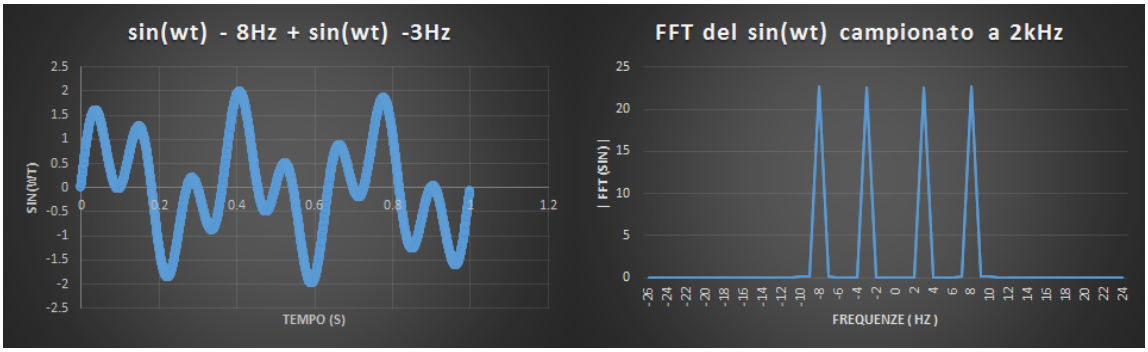

Figura 3.4: Somma delle due sinusoidi precedenti

Come si può notare, abbiamo ritrovato le singole frequenze costituenti il nostro segnale. Vale la pena soffermarsi su una analisi delle singole parti reali e immaginarie della soluzione anziché sul modulo del complesso; a tal proposito, osserviamo il seguente grafico riferito al precedente:

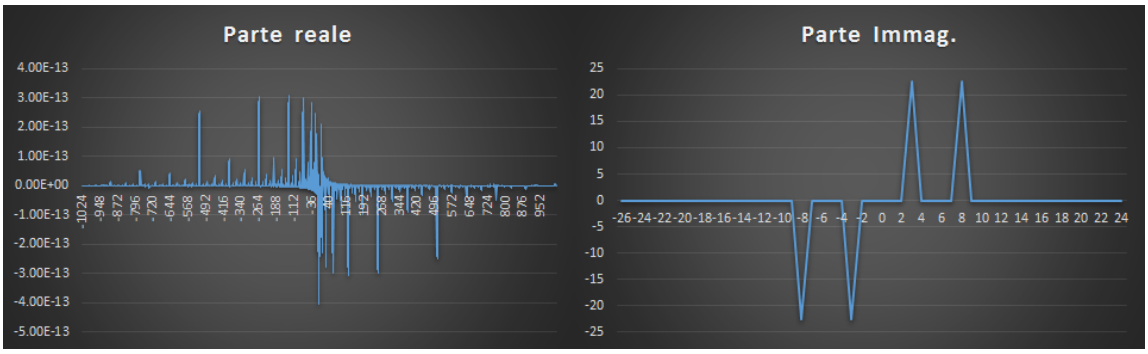

Figura 3.5: Parti reali e immaginarie separate di due sinusoidi (a 8Hz e 3Hz)

Quello che ci aspettiamo (e che abbiamo trovato), trattandosi di una somma tra sinusoidi (parte immaginaria di un esponenziale), è una parte reale identicamente nulla e una parte immaginaria con le frequenze componenti antisimmetriche. Introduciamo, a tal proposito, una trattazione sugli errori e sulla loro propagazione. L'errore più grande che commettiamo in tutta questa argomentazione è un errore di troncamento, dovuto in maggior parte al fatto di aver trasformato un integrale da $-\infty$ a $+\infty$ in una somma discreta entro un intervallo finito di valori, e in minor parte dovuta alle approssimazioni utilizzate nel codice C++, come il valore non esatto di π. Si presuppone quindi che conviene sempre prendere un alto numero di samples, affinché questi errori vengano minimizzati. In ogni caso possiamo considerare i risultati inseriti nei grafici decisamente in accordo con quanto previsto poiché la parte reale ha un valore che si avvicina ai limiti di precisione del calcolatore stesso. Altri due risultati importanti in termini di trasformata di Fourier sono la trasformata di

un onda quadra e la trasformata di una distribuzione di Gauss (che ha ancora una forma a campana come quella di una distribuzione di Gauss con una larghezza, in modulo, più piccola). Allora costruendo un onda quadra di $4Hz$ otteniamo dal nostro programma

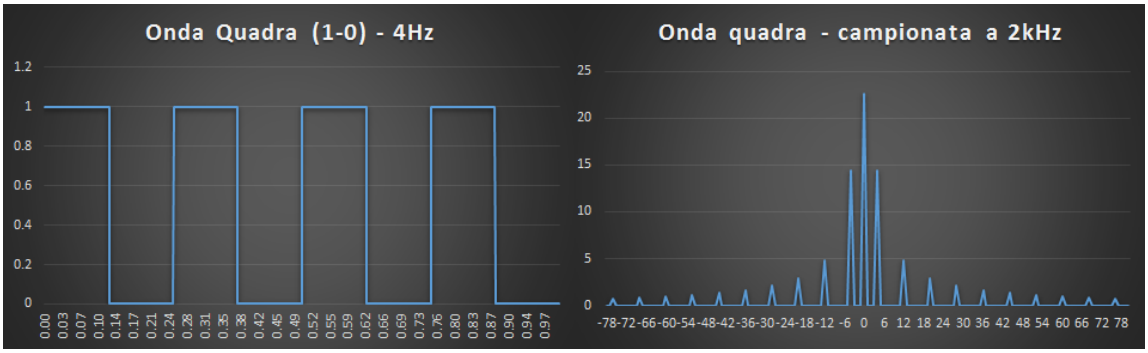

Figura 3.6: Onda quadra e sua trasformata. Nota lo stato alto 1 e basso 0

Come sappiamo l'onda quadra è esprimibile tramite serie di Fourier come somma di armoniche (sinusoidi); tali armoniche infatti sono proprio i picchi successivi al primo che, man mano ci si avvicina alla frequenza di campionamento, vanno velocemente a zero. Parlando invece della trasformata della distribuzione di Gauss abbiamo detto che questa è ancora una Gaussiana con una larghezza dimezzata. Infatti abbiamo:

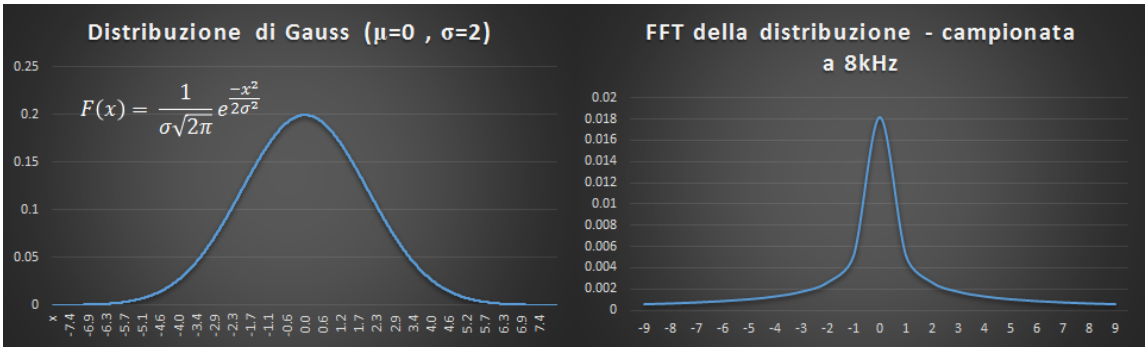

Figura 3.7: Distribuzione di Gauss e la sua trasformata

Conclusioni

La conclusione più degna di nota è la grande potenza del C++, la quale è riassunta in appena 120 righe di codice, facilmente implementabili in un unico oggetto capace di essere semplicemente richiamato in un comando. In questo modo il suo utilizzo si è espanso in ogni direzione, raggiungendo, oggi, limiti mai visti prima. Oggi, infatti, l'algoritmo di fattorizzazione della FT è implementato in una miriade di applicazioni che vanno dallo scopo ludico (programmi di editing audio o foto, cd musicali ecc...), a quello industriale (come le telecomunicazioni) o medico (imaging digitale per risonanze magnetiche e tomografie assiali computerizzate). Esistono versioni modificate di tale algoritmo in grado di svolgere in maniera molto più efficiente casi più specifici come funzioni soltanto reali, o semplicemente funzioni note come sinusoidi o cosinusoidi; ottimizzando in tal senso il codice (in base al tipo di funzione in ingresso appunto), si semplifica ancora di più il numero di calcoli da svolgere e soprattutto si rendono minimi i tempi impiegati, permettendo un rapidissimo responso in termini di frequenza sulla composizione di un segnale; questo che abbiamo descritto rappresenta, in ogni caso, il più generale degli algoritmi. Un ultimo commento va fatto riguardo la dimensionalità della FFT. Noi abbiamo trattato il caso monodimensionale, ma il concetto è facilmente estendibile a più dimensioni; l'utilizzo multidimensionale più sfruttato è quello del caso bidimensionale poichè permette una FFT di intere "matrici" di bit o, forse con il loro termine più noto, pixels.

Bibliografia

[1]William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery - *Numerical Recipes (The Art of Scientific Computing)* - (Cambridge Press, 2007)
[2]F. Ortolani - *Appunti di metodi matematici per la fisica*
[3]Cooley, James W.; Tukey, John W. - *"An algorithm for the machine calculation of complex Fourier series"* - (1965) Math. Comput. 19: 297:301
[4]Nussbaumer - *H.J. 1982, Fast Fourier Transform and Convolution Algorithms (New York: Springer)*