Calcolo della Trasformata di Fourier nei Campi Finiti

Carrer Marco

In questo elaborato viene analizzato il metodo proposto da F. P. Preparata e D. V. Sarwate [6] per il calcolo della trasformata di Fourier discreta (DFT) nei campi finiti. Tale metodo consiste nel convertire il calcolo della DFT nel campo finito in una convoluzione, sempre nel campo finito. La convoluzione viene poi calcolata attraverso una trasformata di Fourier bidimensionale nel campo complesso. L'analisi si sofferma sull'integrazione di queste due procedure in una sola. È stato inoltre realizzato un programma che implementa il metodo ed esegue la DFT di un vettore a coefficienti nel campo finito, e sono state eseguite delle prove per verificarne la correttezza.

Indice

Capitolo 1

Introduzione

1.1 Trasformata di Fourier e Convoluzione

La Trasformata di Fourier é una trasformata integrale di importanza fondamentale in molte applicazioni, come in teoria dei segnali, nell'analisi in frequenza di sistemi dinamici e nella risoluzione di equazioni differenziali. In ambito informatico é noto come, attraverso la trasformata, sia possibile ridurre delle convoluzioni a semplici moltiplicazioni fra fattori. Si utilizza per questo la DFT, ovvero la versione discreta della trasformata di Fourier. Ad esempio, nel campo complesso, essa é definita come:

$$X_k = \sum_{j=0}^{N-1} x_j e^{-\frac{2\pi i}{N} kj}$$

con $k = 0, ..., N-1$, dove X_k, vettore di N numeri complessi $\langle X_0, ..., X_{N-1} \rangle$ é la trasformata di Fourier discreta del vettore x, vettore di N coordinate $\langle x_0, ..., x_{N-1} \rangle$. Nel campo complesso, $e^{\frac{-2\pi i}{N}}$ é una radice primitiva N-esima dell'unitá e per i si intende l'unitá immaginaria. L'antitrasformata é definita quindi come:

$$X_j = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kj}$$

Ora, denotando con $\mathcal{F}$ l'operazione di trasformata e con $*$ l'operazione di convoluzione, possiamo scrivere (senza dimostrarlo):

$$\mathcal{F}(f(x) * h(x)) = \mathcal{F}(f(x)) \cdot \mathcal{F}(h(x))$$

Dove $\cdot$ indica la moltiplicazione fra polinomi (prodotto alla *Cauchy*). Questo fondamentale risultato é noto come **Teorema della convoluzione**: *La trasformata della convoluzione di due funzioni é il prodotto delle trasformate delle due funzioni.* Utilizzando un algoritmo per la trasformata veloce di Fourier (FFT), é possibile trasformare, moltiplicare ed antitrasformare

ottenendo il risultato della convoluzione piú velocemente rispetto al metodo della definizione. Per questo la trasformata é molto utilizzata in informatica, ricoprendo un ruolo fondamentale anche in questo campo.

Questo elaborato, interamente basato sulla pubblicazione *Computational Complexity of Fourier Transform Over Finite Fields* di F. P. Preparata e D. V. Sarwate, ha lo scopo di implementare, comprendendone i dettagli, l'algoritmo proposto in questa pubblicazione, cioé un metodo per effettuare la Trasformata di Fourier in campi finiti, con l'intento di avere un algoritmo universalmente applicabile ed efficiente. L'idea principale é quella di convertire il calcolo della DFT di una sequenza di lunghezza n nel calcolo di una convoluzione ciclica di due sequenze opportunamente definite.

1.2 Campi Finiti

Un Campo soddisfa, con le operazioni somma e prodotto, gli assiomi di anello; inoltre esso é commutativo rispetto al prodotto, ed ogni elemento non nullo ammette inverso, sempre rispetto al prodotto. Le due operazioni godono quindi delle propietá associativa e commutativa ed il prodotto é commutativo rispetto alla somma.

Un Campo Finito (o GF(p^m), campo di Galois con p^m elementi) é un campo caratterizzato da un numero finito di elementi. L'algebra ci insegna che, preso un anello commutativo unitario A, e trovato un elemento p, tale che l'ideale principale (p) sia massimale, considerando le classi di resto $A/(p)$, si ha che $A/(p)$ soddisfa gli assiomi di campo con le operzioni di A. Inoltre $A/(p)$ ha un numero finito di elementi, ed esattamente p. Ora, identificando questo campo con A' e considerando il suo anello di polinomi $A'[x]$, si puó applicare lo stesso ragionamento: trovato un elemento $g(x)$ in $A'[x]$ tale che l'ideale $(g(x))$ sia massimale, si ha che $A'[x]/(g(x)) \equiv \mathbb{K}$ con $\mathbb{K}$ campo. Se m é il grado del polinomio $g(x)$, il campo trovato avrá ovviamente p^m elementi

Partendo dall'insieme dei numeri interi $\mathbb{Z}$, che é anello commutativo con le operazioni somma e prodotto, é possibile ottenere un campo con p elementi semplicemente considerando $\mathbb{Z}/p\mathbb{Z}$ con p elemento primo. Infatti essendo $\mathbb{Z}$ anello fattoriale si ha che ogni elemento primo é irriducibile e massimale. Quindi $\mathbb{Z}/p\mathbb{Z}$ é campo, e, sapendo che l'anello di polinomi di un anello fattoriale é anch'esso fattoriale, si puó ripetere il ragionamento precedente; conoscendo infatti un polinomio $g(x)$ appartente all'anello dei polinomi di $\mathbb{Z}/p\mathbb{Z}$, di grado m ed irriducibile, si effettua come visto prima il modulo, ottenendo un campo con p^m elementi, quindi una rappresentazione di GF(p^m). I valori che identificano il campo GF(p^m) sono quindi p ed m, oltre al polinomio irriducibile $g(x)$, fondamentale per costruire il campo stesso e poter svolgere al suo interno le operazioni di somma e prodotto. [7] Ma trovare un polinomio irriducibile del grado voluto non é un problema banale; non esiste infatti un metodo universale per trovare tali polinomi, che andrebbero calcolati escludendo dall'insieme di polinomi di grado m quelli divisibili per un polinomio di grado minore di m.

1.3 Rappresentazione elementi in GF(p^m)

Ogni elemento $a(x)$ appartenente a GF(p^m) é rappresentato, da quanto detto prima, come un polinomio (classi resto dell'anello di polinomi in GF(p^m) modulo g(x)) in questo modo:

$$a(x) = \sum_{i=0}^{m-1} a_i x^i \tag{1.1}$$

poiché ha grado massimo $m-1$. Nel calcolatore questa rappresentazione puó avvenire in maniera semplice attraverso l'utilizzo di *array* in questo modo:

$$a_0 + a_1 x + a_2 x^2 + ... + a_{m-1} x^{m-1} \rightarrow \langle a_0, a_1, a_2, ..., a_{m-1} \rangle$$

dove quindi il coefficente a_i viene memorizzato nella i-esima posizione di un array di dimensione m. Questa rappresentazione é efficace in quanto ogni polinomio in GF(p^m) é identificato univocamente, come elemento del campo, dai coefficenti di $x^0, x^1, ...x^{m-1}$, i quali appartengono a GF(p).

Capitolo 2

Studio Teorico

2.1 La tecnica di *Bluestein*

Sia a un vettore di lunghezza n di elementi in GF(p^m). La componente j-esima A_j della trasformata (DFT) del vettore a é definita come:

$$A_j = \sum_{i=0}^{n-1} a_i \alpha^{ij}$$

con α radice n-esima dell'unitá in GF(p^m). La tecnica di *Bluestein* si basa sulla seguente espressione:

$$\sum_{i=0}^{n-1} a_i \alpha^{ij} = \sum_{i=0}^{n-1} a_i \alpha^{1/2[i^2+j^2-(j-i)^2]} = \beta^{j^2} \sum_{i=0}^{n-1} (a_i \beta^{i^2})(\beta^{-(j-i)^2})$$

con $\beta = \sqrt{\alpha}$. Questa espressione ripensa il prodotto ij come metá del doppio prodotto derivante dal quadrato di $(j-i)$, o meglio:

$$(j-i)^2 = j^2 - 2ij + i^2 \Rightarrow ji = \frac{1}{2}(j^2 + i^2 - (j-i))$$

In questo modo si puó riconoscere nell'ultima espressione della sommatoria precedente la convoluzione ciclica di due sequenze. Indicando con la notazione $(a_i)_k^l(0)^t$ (con k<l<t) il vettore $\langle a_k, a_{k+1}, ..., a_l, 0, ..., 0\rangle$ con t zeri finali, possiamo identificare i due vettori da convolvere come $(a_i\beta^{i^2})_0^{n-1}(0)^{N-n}$ e $(\beta^{-i^2})_{1-n}^{n-1}(0)^{N-(2n-1)}$, dove $N \geq 2n-1$. Questa tecnica é molto utilizzata nella trasformata veloce di Fourier perché riesprime la DFT come una convoluzione, la cui dimensione puó essere portata ad una potenza di due e valutata dalla FFT con tecniche diverse, ad esempio *Cooley-Tukey*.

2.2 Radici dell'unitá in GF(p^m)

Nel paragrafo precedente é stata introdotta β come radice $2n$-esima dell'unitá in GF(p^m), ma l'esistenza di questa in GF(p^m) non é scontata. Dalla teoria sappiamo infatti che in GF(p^m) esiste una radice h-esima se e solo se $h \mid (p^m - 1)$. Distinguiamo allora tre casi:

- $p = 2$
- $p \neq 2$ ma $2n \mid (p^m - 1)$
- $p \neq 2$ e $2n \nmid (p^m - 1)$

Sempre dalla teoria ricordiamo che, se α é radice primitiva n-esima dell'unitá, ovviamente si ha $\alpha^n = 1$ e quindi $\alpha^{n+1} = \alpha$. Ora, sappiamo che $n \mid (p^m - 1)$ (questo é provato dall'esistenza di α [5]), quindi, nel caso in cui $p = 2$, $p^m - 1$ deve essere dispari, cosí come n. Allora $n + 1$ é pari e si puó scrivere l'uguaglianza $\beta = \sqrt{\alpha} = \alpha^{1/2(n+1)}$, poiché $(n + 1)/2$ é intero. In questo caso abbiamo quindi un semplice modo per calolare la radice $2n$-esima.

Nel secondo caso l'esistenza della radice $2n$-esima é assicurata dal fatto che $2n$ divide $(p^m - 1)$, anche se non si dispone di un modo semplice per calcolarla; in questo caso verrá calcolata elevando iterativamente ogni elemento del campo alla seconda e confrontando il risultato con α fino a trovare la radice cercata.

Nel terzo caso invece non esiste la radice $2n$-esima in GF(p^m), quindi per ricercarla bisogna immergersi in GF(p^{2m}), dove l'esistenza di tale radice é garantita. Per fare questo dobbiamo trovare un polinomio irriducibile di grado $2m$ (sia questo $h(x)$), considerare gli elementi del campo GF(p^{2m}) come le classi resto dell'anello di polinomi in GF(p) modulo $h(x)$ ed infine definire un applicazione iniettiva $\varphi : GF(p^m) \longrightarrow GF(p^{2m})$, per identificare ogni elemento di GF(p^m) con un elemento in GF(p^{2m}) in modo da, una volta svolti i conti necessari, ritornare attraverso φ^{-1} al campo di partenza. Come giá affermato in precedenza, il solo calcolo di un polinomio irriducibile di grado voluto non é affatto banale, inoltre anche la definizione dell'applicazione φ introduce notevoli difficoltá.

2.3 Trasformare polinomi

Utilizzando per gli elementi di GF(p^m) la rappresentazione 1.1, vediamo ora il calcolo della convoluzione ciclica di a ed b, due vettori (di lunghezza n) di elementi GF(p^m). Se $(w_j)_0^{N-1}$ denota il risultato di tale convoluzione, identificando con $a_{i,k}$ il k-esimo coefficente del polinomio che rappresenta $a_i \in \mathrm{GF}(p^m)$, j-esima componente del vettore a, si ha:

$$\begin{aligned} w_j(x) &= \sum_{i=0}^{N-1} \left(a_i(x) b_{((j-i))}(x)\right) \bmod g(x) \\ &= \sum_{i=0}^{N-1} \left(\sum_{t=0}^{2m-2} \left(\sum_{k=0}^{t} a_{i,k} b_{((j-i)),t-k} \right) x^t \right) \bmod g(x) \\ &= \sum_{t=0}^{2m-2} \left(\sum_{k=0}^{t} \left(\sum_{i=0}^{N-1} a_{i,k} b_{((j-i)),t-k} \right) x^t \right) \bmod g(x) \end{aligned}$$

dove $a_{i,k}$ e $b_{((j-i)),t-k}$ sono zero se k o $t-k$ risultano maggiori di $m-1$, e $((j-i))$ indica l'operazione $(j-i) \bmod N$. Ora, definendo il numero intero $\bar{z}_{j,t}$ come:

$$\bar{z}_{j,t} \triangleq \sum_{k=0}^{t} \left(\sum_{i=0}^{N-1} a_{i,k} b_{((j-i)),t-k} \right)$$

con $j \in [0, N-1]$ e $t \in [0, 2m-2]$ si ha che:

$$\bar{z}_{j,t} \bmod p = \sum_{k=0}^{t} \left(\sum_{i=0}^{N-1} a_{i,k} b_{((j-i)),t-k} \right) \in GF(p)$$

Questo ci permette di pensare la convoluzione precedente come convoluzioni di coefficenti interi e non in GF(p), evitando di eseguire il modulo ad ogni operazione. Osservando la defininizione di $\bar{z}_{j,t}$ si puó riconoscere una convoluzione a due dimensioni, periodica in una dimensione ed aperiodica nell'altra. Quest'ultima puó essere facilmente trasformata in una convoluzione periodica estendendo la dimensione di k da $[0,t]$ a $[0, M-1]$ con $M \geq 2m-1$. Quindi, utilizzando la notazione $[[t-k]] = (t-k) \bmod M$, $\bar{z}_{j,t}$ puó essere riscritto come:

$$\bar{z}_{j,t} = \sum_{k=0}^{M-1} \left(\sum_{i=0}^{N-1} a_{i,k} b_{((j-i)),[[t-k]]} \right)$$

Conviene per questo pensare $a_{i,k}$, con $0 \leq i \leq N-1$ e $0 \leq k \leq M-1$, come un array bidimensionale e $\bar{z}_{j,t}$ come una convoluzione di array.

Definiamo ora la trasformata di Fourier bidimensionale di $a_{i,k}$ nel campo dei numeri complessi in questo modo:

$$A_{j,l} = \sum_{k=0}^{M-1} \sum_{i=0}^{N-1} a_{i,k} \Omega_N^{ij} \Omega_M^{kl}$$

dove $\Omega_N = e^{j2\pi/N}$ e $\Omega_N = e^{j2\pi/M}$ sono radici dell'unitá rispettivamente N-esime ed M-esime, nel campo complesso. Allora, grazie a questa trasformata, possiamo scrivere:

$$\bar{Z}_{j,t} = X_{j,t} B_{j,t}$$

dove $\bar{Z}_{j,l}$ e $B_{j,l}$ indicano la trasformata di Fourier bidimensionale degli array $\bar{z}_{i,k}$ e $b_{i,k}$. É utile osservare che, ridefinendo la trasformata di Fourier bidimensionale come due trasformate in due diverse dimensioni (righe e colonne) si puó utilizzare la trasformata delle righe per moltiplicare efficacemente i polinomi. Infatti le righe di un vettore di elementi di GF(p^m) non sono altro che polinomi di grado $m-1$; se questi vengono trasformati si puó utilizzare il teorema di convoluzione per ridurre la moltiplicazione di due polinomi, vista come una convoluzione, ad una moltiplicazione coefficente per coefficente.

Ricordando poi che una delle sequenze da convolvere é $(a_i \beta^{i^2})_0^{n-1}(0)^{N-n}$, questa risulta essere il prodotto, termine per termine, delle due sequenze $(a_i)_0^{n-1}(0)^{N-n}$ e $(\beta^{i^2})_0^{n-1}(0)^{N-n}$; quindi, rappresentando quest'ultime come polinomi di grado $m-1$, come visto prima, il loro prodotto sará un polinomio di grado $2m-2$. L'operazione di convoluzione con $(\beta^{-i^2})_{1-n}^{n-1}(0)^{N-(2n-1)}$ e la successiva moltiplicazione per β^{j^2} dará un polinomio di grado $4m-4$. É facile dimostrare che effettuare il modulo $g(x)$ alla fine di queste operazioni dá lo stesso risultato che effettuarlo alla fine di ogni operazione, facendo risparmiare tempo all'algoritmo. Per questo motivo allora dobbiamo scegliere M tale da contenere un polinomio di grado $4m-4$. Gli array avranno infatti dimensione, NxM, dove, ricapitolando, $N \geq 2n-1$ ed $M \geq 4m-4$. In vista del calcolo della DFT mediante un algoritmo veloce (FFT), scegliamo $N = 2^s$ ed $M = 2^r$, con $s = \log_2(2n-1)$ ed $r = \log_2(4m-4)$.

Capitolo 3

Implementazione

3.1 Algoritmo

Grazie a quanto affermato in precedenza, possiamo qui riportare i dati di cui necessita l'algoritmo. Definiamo gli array NxM $[a_{i,k}]$, $[b_{i,k}]$ ed $[y_{i,k}]$ che contengono rispettivamente le sequenze $(a_i)_0^{n-1}(0)^{N-n}$, $(\beta^{i^2})_0^{n-1}(0)^{N-n}$ e $(\beta^{-i^2})_{1-n}^{n-1}(0)^{N-2n+1}$. In questo modo, la sequenza $(a_i\beta^{i^2})_0^{n-1}(0)^{N-n}$ puó essere ricavata moltiplicando tra loro le righe (moltiplicazione fra polinomi) delle sequenze $[a_{i,k}]$ e $[a_{i,k}]$, $[b_{i,k}]$. Cosí come la moltiplicazione finale di ogni componente della trasformata per β^{j^2}, richiesta da *Bluestein*, puó avvenire fra la sequenza risultato della convoluzione e la sequenza $[b_{i,k}]$, cambiando opportunamente gli indici, dato che le componenti che ci interessano della trasformata si presentano dalla $(n-1)$-esima posizione in poi. Queste andranno quindi moltiplicate a partire dalla riga 0 della sequenza $[b_{i,k}]$ e non dalla $n-1$-esima.

Sempre per quanto detto prima, possiamo utilizzare la trasformata di Fourier per semplificare queste, ed altre operazioni. Definiamo quindi la *Row Fourier Transform* dell'array $[x_{i,k}]$ (o RFT$[x_{i,k}]$) come l'array $[x'_{i,l}]$, dove:

$$x'_{i,l} = \sum_{k=0}^{M-1} x_{i,k}\Omega_M^{kl}$$

e la *Column Fourier Transform* dello stesso *array* (o CFT$[x_{i,k}]$) come l'*array* $[x''_{i,k}]$ dove:

$$x''_{j,l} = \sum_{i=0}^{N-1} x_{i,k}\Omega_N^{ij}$$

É facile verificare che CFT[RFT$[x_{i,k}]$]=RFT[CFT$[x_{i,k}]$]=CRFT$[x_{i,k}]$=$[X_{j,l}]$, dove $[X_{j,l}]$ é la trasformata di Fourier bidimensionale introdotta nel precendente paragrafo. Ora abbiamo tutti i mezzi per introdurre l'algoritmo.

Algoritmo:

Step 1. $[a'_{i,l}] \longleftarrow \mathrm{RFT}[a_{i,k}]$, $[b'_{i,l}] \longleftarrow \mathrm{RFT}[b_{i,k}]$
Step 2. $[x'_{i,l}] \longleftarrow [a'_{i,l} \cdot b'_{i,l}]$
Commento: le righe di $[x'_{i,l}]$ rappresentano la trasformata di $a_i\beta^{i^2}$.
Step 3. $[X_{j,l}] \longleftarrow \mathrm{CFT}[x'_{i,l}]$
Step 4. $[Y_{j,l}] \longleftarrow \mathrm{CRFT}[y'_{i,k}]$
Step 5. $[\bar{Z}_{j,l}] \longleftarrow [X_{j,l} \cdot Y_{j,l}]$
Commento: $[\bar{Z}_{j,l}]$ rappresenta la trasformata della convoluzione derivante dall'utilizzo della tecnica di *Bluestein*
Step 6. $[\bar{z}'_{i,l}] \longleftarrow \mathrm{CFT}^{-1}[\bar{Z}_{j,l}]$
Step 7. $[\bar{u}'_{i,l}] \longleftarrow [\bar{z}'_{i,l} \cdot b'_{[[i-(n-1)]],l}]$
Commento: nel passaggio precedente viene effettuata la moltiplicazione fra il risultato della convoluzione dovuta a *Bluestein* e β^{i^2}
Step 8. $[u_{i,k}] \longleftarrow \mathrm{RFT}^{-1}[\bar{u}'_{i,l}]$
Commento: $[u_{i,k}]$ é la trasformata di Fourier cercata, rappresentata con polinomi di grado $4m-4$
Step 9. $[u_{i,k}] \longleftarrow [\bar{u}_{i,k} \bmod p]$
Step 10. $A_i \longleftarrow \left(\sum_{k=0}^{4m-4} u_{i,k}x^k\right) \bmod g(x)$ per i=0,1,...,$n-1$

3.2 Programma

Il programma realizzato segue passo passo lo schema dell'algoritmo precedente. Per definire il campo sul quale operare, é necessario ovviamente inserire i dati che lo caratterizzano, quindi p, m ed il polinomio irriducibile di grado m, oltre a n oltre ed alla radice primitiva n-esima dell'unitá. Si é scelto di rappresentare i polinomi con un array di dimensione M, contenente i coefficenti del polinomio stesso. Questi sono memorizzati dal meno significativo, alla posizione 0 dell'array, al piú significativo, nella posizione corrispondente al grado del polinomio rappresentato, che puó essere maggiore di m, come visto in precedenza, ma non puó essere maggiore di $4m-4 \leq M$. I coefficenti dei polinomi appartengono a GF(p) e sono quindi numeri interi; si sarebbe potuto scegliere quindi di utilizzare array di *int*, invece si é preferito utilizzare un tipo di dato che permettesse la rappresentazione di numeri in virgola mobile. Questo in vista di svolgere le operazioni di trasformata di Fourier delle righe della matrice. Questa trasformata viene infatti calcolata nel campo complesso, utilizzando l'apposita libreria *complex*, che fornisce i metodi per operare in tale campo, nonché quelli per passare da una notazione polare ad una notazione cartesiana. In particolare la trasformata opera con le radici primitive M-esime dell'unitá, che sono rappresentate come numeri complessi con componenti (reale e immaginaria) in virgola mobile. Dato quindi che gli array vengono trasformati ed antitrasformati, si é preferito utilizzare subito una rappresentazione che non obbligasse ad effettuare *cast* di tipi di dati. Lo stesso discorso puó essere fatto per quanto riguarda le matrici contenenti il vettore da trasformare e i risultati intermedi, dato che queste sono sequenze di N polinomi e che ad esse viene applicata la trasformata di Fourier delle colonne, calcolata sempre nel campo complesso utilizzando le radici primitive N-esime dell'unitá. Detto questo, gli array che rappresentano i polinomi (e le matrici), sono definiti come array di *double*, e di conseguenza i numeri complessi sono stati definiti come *complex<double>*.

Qui sotto viene esposto in breve lo scopo delle funzioni principali realizzate.

ModuloP:

> Questa funzione effettua il modulo p di un polinomio (quindi un array di dimensione M) passatogli come argomento. Realizza questo semplicemente con un ciclo che, da 0 ad M, utilizza il metodo *fmod* della libreria *cmath* che calcola appunto il resto della divisione fra i due valori passatigli come argomento.

calcolaX

> Per calolare il modulo $g(x)$ dei polinomi c'é bisogno di calcolare le potenze $x^m, x^{m+1}, ..., x^{M-1}$ come polinomi in GF(p^m). Ques-

ta funzione popola una matrice di dimensioni $(M - m + 1)$xm inserendo, ad ogni riga i-esima, il polinomio corrispondente a x^{m+i}. Per fare questo per prima cosa calcola x^m calcolando l'inverso rispetto alla somma in GF(p) dei coefficenti del polinomio irriducibile, sapendo che l'inverso di g_j, coefficente j-esimo di $g(x)$, é $g'_j = p - g_j$. Successivamente calcola i coefficenti $g_{i,j}$, coefficente j-esimo del polinomio corrispondente a x^{m+i}, come:

$$g_{i,0} = g_{i-1,m-1} \cdot g_{i-1,0}$$

e per $j > 0$

$$g_{i,j} = g_{i-1,m-1} \cdot g_{i-1,j} + g_{i-1,j-1}$$

Modulo:

Conoscendo le potenze di x maggiori di m, calcolate con la funzione precedente, per ricavare il modulo $g(x)$ di un polinomio $a(x)$ questo metodo deve solamente sommare ad a_j i coefficenti $g_{i,j}$ del polinomio corrispondente a x^{m+i} per $m \leq i \leq t$ con t grado del polinomio $a(x)$ passato al metodo come argomento. Prima di restituire il polinomio cosí ottenuto, ne effettua il *moduloP*.

Moltiplica:

Metodo che moltiplica i due polinomi passatigli come argomento, utilizzando il prodotto alla *Cauchy*. Restituisce quindi il prodotto, senza effettuarne il modulo $g(x)$, quindi in uscita é possibile avere anche polinomi di grado maggiore di m. Questo metodo non é utilizzato all'interno dell'algoritmo della trasformata ma é utile per calcolare le potenze degli elementi in GF(p^m), oltre che per ricercare l'inverso di un altro elemento.

Inverso:

Ricerca iterativamente (passando in rassegna tutti gli elementi di GF(p^m)) l'elemento inverso dell'elemento $a(x)$ passatogli come argomento. Per farlo utilizza *Moltiplica* per moltiplicare appunto ogni polinomio in GF(p^m) per $a(x)$, confrontando poi il prodotto con elemento *uno* ($\langle 1, 0, 0, ..., 0 \rangle$). Restituisce quindi l'elemento trovato, la cui esistenza é garantita dal fatto che si sta operando in un campo.

RowFT, ColumnFT, aRowFT, aColumnFT:

Questi metodi realizzano la trasformata e l'antitrasformata di Fourier nel campo complesso di righe e colonne, esattamente

come descritta nel paragrafo precedente. Non sono state infatti adottate tecniche di trasfomata veloce di Fourier che avrebbero migliorato le performance dell'algoritmo, poiché lo scopo era quello di realizzare questo comprendendone il funzionamento.

Main:

All'avvio il programma richiede i dati necessari per operare, nell'ordine:

- il numero primo p (ed effettua un controllo per verificare sia primo);
- il numero m, grado del polinomio irriducibile (che caratterizza il campo);
- il numero n, cardinalitá della trasformata (ed effettua il controllo per verificare che n divida $p^m - 1$);
- i coefficienti del polinomio irriducibile $g(x)$, dal meno al piú significativo;
- i coefficienti del polinomio $\alpha(x)$ radice primitiva n-esima dell'unitá, sempre dal meno al piú significativo (ed effettua il controllo per verificare che sia radice dell'unitá e che sia radice primitiva).

Inseriti questi dati il programma passa al calcolo di β distinguendo i tre casi distinti anche nel paragrafo 2.1. Se $p = 2$ allora $\beta = \alpha^{(n+1)/2}$; se $p \neq 2$ e $2n \mid (p^m - 1)$ allora prova ad elevare alla seconda ogni elemento di GF(p^m), fermandosi quando trova l'elemento che elevato alla seconda restituisce α, cioé il β cercato. Se invece $p \neq 2$ e $2n \nmid (p^m - 1)$, abbiamo visto in precedenza come questo caso complichi molto il calcolo della radice cercata, poiché per immergersi in GF(p^m) avremmo bisogno di un polinomio irriducibile di grado $2m$; per questo motivo si é deciso di non far proseguire il programma se i dati inseriti rispecchiano questo caso. Viene quindi restituito un messaggio che spiega l'impossibilitá a proseguire, dato che non é possibile trovare β. A questo punto tutti i dati sono stati inseriti ed il programma prosegue leggendo una matrice nxm contenuta in un file di testo. Questa corrisponde al vettore da trasformare, quindi le righe di questa matrice corrispondono ai coefficienti degli n polinomi che formano il vettore, coefficienti utilizzati per costruire $[a_{i,k}]$ contenente la sequenza $(a_i)_0^{n-1}(0)^{N-n}$. Poi, servendosi del metodo *Moltiplica*, utilizzato dentro ad un ciclo per calcolare le potenze di β che compongono la sequenza $(\beta^{i^2})_0^{n-1}(0)^{N-n}$, calcola $[b_{i,k}]$ e, utilizzando gli stessi risultati, grazie al metodo *Inverso*

calcola la sequenza $(\beta^{-i^2})_{1-n}^{n-1}(0)^{N-2n+1}$ costruendo la matrice $[y_{i,k}]$. A questo punto, utilizzando le funzioni *RowFT, ColumnFT, aRowFT* ed *aColumnFT*, l'algoritmo segue pari passo gli step descritti in precedenza, ottenendo la trasformata desiderata.

É stato poi realizzato un altro programma che effettua la stessa trasformata di Fourier nel campo finito, ma utilizzando la definizione formale della trasformata, senza nessuna tecnica di miglioramento. Questo ha il solo scopo di confrontare il risultato con quello ottenuto dal programma oggetto della tesina.

3.3 Prove effettuate:

Una volta scritto il programma, per verificarne la correttezza sono state esguite delle prove. In particolar modo si é provato a trasformare un vettore notevole come quello formato da tutte le componenti uguali all'elemento *uno*, che come polinomio é rappresentato dall'array $\langle 1, 0, 0, ..., 0\rangle$. La trasformata che ci si aspetta é un impulso, quindi con tutte le coordinate uguali a *zero* ($\langle 0, 0, 0, ..., 0\rangle$), tranne la prima che deve essere *uno*.
Si é voluto prendere come campo di prova un campo con pochi elementi e la scelta é ricaduta su GF(3^2), quindi con $p = 3$ ed $m = 2$. Si é scelto $p \neq 2$ in modo che il calcolo della radice $2n$-esima dell'unitá fosse svolto con il metodo iterativo, in modo da testare anche questo. Ovviamente si é scelto n in modo che $2n$ dividesse $p^m - 1$, quindi semplicemente $n = 4$. Come polinomio irriducibile di secondo grado é stato scelto:

$$g(x) = 1 + x^2$$

grazie al quale é possibile operare in un campo, mentre come radice primitiva dell'unitá é stata scelta:

$$\alpha(x) = x$$

infatti calcolando x^2 a partire dal polinomio irriducibile si ottiene $x^2 = -1 = (p - 1) = 2$. In GF(3^2) si ha quindi:

$$(\alpha(x))^4 = x^4 = x^2 \cdot x^2 = 2 \cdot 2 = 4 \bmod 3 = 1$$

Avviato il programma ed inseriti i dati, questo restituisce per prima cosa la radice $2n$-esima calcolata $\beta = 2 + x$ che risulta esatta. Si ha infatti:

$$(2 + x)^2 = 4 + 4x + x^2 = 1 + x + 2 = x = \alpha(x)$$

La matrice nxm da trasformare é, come detto prima:

$$a = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{pmatrix}$$

Questa, espansa attraverso l'aggiunta di zeri alla dimensione MxN, risulta:

$$a_{MxN} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

L'uscita u ottenuta dall'algoritmo avrebbe anch'essa dimensione MxN; ma dato che le componenti che ci interessano (righe della matrice), sono le componenti u_i con $n-1 \leq i < 2n-1$, é possibile far svolgere all'algoritmo le ultime operazioni (che coinvolgono coefficenti della matrice appartenenti alla stessa riga) solo alle righe di nostro interesse. Cosí, l'operazione di prodotto della componente u_i, vista come polinomio, con β^{i^2}, e l'operazione $\mathrm{mod}\, g(x)$, possono essere effettuate solo per le righe che andranno a formare la trasformata voluta. Ad esempio, in questa prova, dove $M = N = 8$, queste operazioni vengono svolte solo per le righe u_i, con $3 \leq i < 7$, le stesse righe che andranno a formare la trasformata cercata.

Detto questo, la trasformata di Fourier ottenuta durante la prova coincide con quanto ci si aspettava e vale:

$$u = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Verificato il corretto funzionamento dell'algoritmo, si é passati alla prova con diversi valori di p, m ed n.

Si é provato con $p = 2$, verificando cosí il corretto calcolo di $\beta = \alpha^{(n+1)/2}$, con $m = 2$ ed $n = 3$, utilizzando i polinomio irriducibile e la radice dell'unitá $g(x)_{m=2} = 1 + x + x^2$ ed $\alpha(x)_{m=2} = 1 + x$ (campo di 4 elementi). Poi, sempre con $p = 2$, si é testato il caso $m = 3$ ed $n = 7$, con i polinomi $g(x)_{m=3} = 1 + x + x^3$. e $\alpha(x)_{m=3} = 1 + x$ (campo di 8 elementi).

Sono state fatte poi altre prove con $p = 3$ aumentando m per utilizzare campi con piú elementi:

- con $m = 3$, $n = 13$, $g(x)_{m=3} = 1 + 2x + x^3$ ed $\alpha(x)_{m=3,n=13} = 2 + 2x^2$ (campo con 27 elementi);
- con $m = 4$, $n = 40$, $g(x)_{m=4} = 2+x+x^4$ ed $\alpha(x)_{m=4,n=40} = 1+x^2+x^3$ (campo con 81 elementi);
- con $m = 5$, $n = 121$, $g(x)_{m=5} = 1 + 2x + x^5$ ed $\alpha(x)_{m=5,n=121} = 1 + x^2 + x^4$ (campo con 243 elementi);

- con $m = 6$, $n = 182$, $g(x)_{m=6} = 2 + x + x^6$ ed $\alpha(x)_{m=6,n=182} = 2 + x^2 + x^4 + x^5$ (campo con 729 elementi);
- con $m = 8$, $n = 205$, $g(x)_{m=8} = 2 + x^3 + x^8$ ed $\alpha(x)_{m=8,n=205} = 2 + 2x^3 + x^4 + x^6$ (campo con 6561 elementi);

Durante queste prove é stato trasformato un vettore (creato semplicemente ponendo il coefficente dell'i-esima riga e della j-esima colonna $a_{i,j} = (i + 2j) \bmod p$) utilizzando l'algoritmo visto ed un secondo algoritmo, programmato seguendo la definizione della trasformata di Fourier nel campo finito. Questo con il fine di confrontare i due risultati per verificare la correttezza dell'algoritmo oggetto della tesina.

3.4 Codice

Di seguito viene proposto il codice *C++* di quanto realizzato.

```
//        Fintite Field Fourier Transform
//        algoritmo basato su quanto descritto da
//        F. P. Preparata e D. V. Sarwate

#include <cmath>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <complex>
#define PI 3.14159265358979323
using namespace std;

unsigned int p,m,n;
unsigned int N,M;
bool pm=false;

/*funzione di controllo se il numero e' primo*/

bool primo(unsigned int num){
  for (unsigned int i=2 ;i<=num/2; i++){
    if (num % i==0) return false;
  }
  return true;
}

/*funzione di controllo se n(p^m-1)*/

bool divide(double nn){
  double card=pow(p,m)-1;
  if (fmod(card,nn)==0) return true;
  else return false;
}


/* funzione che calcola la piu' piccola potenza di 2
 * tale che sia maggiore di lim*/

int explog(int lim){
  int i=1;
```

```
  while (pow(2,i)<lim)
    i++;
  return pow(2,i);
}

/*funzione per leggere da file*/

void leggiMatrice(complex<double> *matrice) try{
  ifstream IN("matrice");
  if(!IN) throw(1);
  for (unsigned int i=0; i<N; i++){
    for (unsigned int j=0; j<M; j++){
      if ((i>=m) or (j>=n)) {*(matrice+j+M*i)=0;}
      else {IN>>*(matrice+j+M*i);}
    }
  }
}

catch(int e){
  if(e) cout<<"input non valido\n";
}

/*calcola e restituisce la radice dell'unita' di grado degree*/

complex<double> radice(int degree){
  complex<double> root;
  root=polar((double)(1), 2*PI/degree);
  return root;
}

void stampaMatrice (complex<double> *matrice){
  cout << endl;
  for (unsigned int i=0; i<N; i++){
    for (unsigned int j=0; j<M; j++){
      cout << matrice[(M*i)+j] << " ";
    }
  cout << endl;
  }
}

void stampaMatrice (double *matrice){
  cout << endl;
  for (unsigned int i=0; i<n; i++){
    for (unsigned int j=0; j<m; j++){
```

```
      cout << matrice[(m*i)+j] << " ";
    }
  cout << endl;
  }
}

/* calcola il grado di un polinomio*/

unsigned int grado(double* poly){
  unsigned int g=M-1;
  while((poly[g]==0)and(g>0)){
    g--;}
  return g;
}

/* confronta 2 polinomi per stabilire se sono ugugali*/

bool uguali(double * poly1, double * poly2){
  int g1, g2;
  g1=grado(poly1);
  g2=grado(poly2);
  if (g1!=g2)
    return false;
  else{
    for (int i=0;i<=g1;i++){
      if (poly1[i]!=poly2[i])
        return false;
    }
  return true;
  }
}

void stampaPolinomio(double* poly, int g){
  for (int i=0;i<=g;i++){
    cout <<poly[i]<<"*"<< "x^"<<i;
    if (i!=g)
      cout <<"+";
  }
  cout << endl;
}

/* funzione moltiplica:
 * moltiplica 2 elementi appartenti a GF(p^m) ritornando
 * come risultato un polinomio non in GF(p^m), quindi di
```

```
 * grado anche > m (ma < di 4n-4) e a coefficenti interi
 * (prodotto alla Cauchy)*/

double * moltiplica(double* f1, double* f2){
  double * r = new double[M];
  for (unsigned int i=0; i< M; i++){
    r[i]=0;
    for (unsigned int j=0; j<=i; j++){
      r[i]=r[i]+(f1[i-j]*f2[j]);
    }
  }
  return r;
}

/* funzione modulo:
 * datogli un polinomio in ingresso, ne esegue il modulo
 * g(x) in GF(p^m), con g(x) polinomio irriduibile a
 * coefficenti in GF(p)*/

void moduloP(double* poly){
  for (unsigned int i=0; i<M; i++){
    if (poly[i]>=p){
      poly[i]=fmod(poly[i],(double)(p));
    }
  }
}

void calcoloX(double* irr, double* x){
  double* check = new double[m];                    //
  int amonic=1;
  if (irr[m]>1){
    for(int am=0;am<p;am++){
      if (((int)(irr[m])*am)%p==1){
        amonic=am;
        break;
      }
    }
  }
  for (unsigned int i=0; i<m;i++){
    if(irr[i]==0)
      x[i]=0;
    else
      x[i]=(p-irr[i])*amonic;
  }
```

```
  for (unsigned int j=1;j<M-m+1;j++){
    x[j*m]=x[(j-1)*m+(m-1)]*x[0];
    for(unsigned int s=1;s<m;s++){
      x[j*m+s]=x[(j-1)*m+(m-1)]*x[s]+x[(j-1)*m+(s-1)];
    }
    for (unsigned int z=0; z<m; z++){
      check[z]=x[j*m+z];
        }
    moduloP(check);
    for (unsigned int zz=0;zz<m;zz++){
      x[j*m+zz]=check[zz];
    }
  }
}

void modulo(double* poly, double* x){
  moduloP(poly);
  unsigned int g = grado(poly);
  for (unsigned int l=g;l>=m;l--){
    for (unsigned int t=0;t<m;t++){
      poly[t]=poly[t]+x[(l-m)*m+t]*poly[l];
    }
    poly[l]=0;
  }
  moduloP(poly);
}

/* funzione che calcola e restituisce l'elemento
 * inverso dell'elemento passato come argomento,
 * nel campo finito*/

double * inverso(double *arg, double *xp){
  int ii=0;
  double * element = new double[M];
  double * buff = new double[M];
  double * uno = new double[M];
  for (int j=0;j<M;j++){
    element[j]=0;
    uno[j]=0;
  }
  if (p==2)
    element[1]=1;
  else
    element[0]=2;
```

```
  uno[0]=1;
  while (element[m-1]<p){
    buff = moltiplica(element,arg);
    if (grado(buff)>=m)
      modulo(buff, xp);
    else
      moduloP(buff);
    if (uguali(buff,uno)){
    delete [] buff;
    delete [] uno;
      return element;
    }
    element[0]=element[0]+1;
    while ((element[ii]==p)and(ii<m-1)){
      element[ii]=0;
      element[ii+1]=element[ii+1]+1;
      ii++;
    }
    ii=0;
  }
  delete [] buff;
  delete [] element;
  return uno;
}

/* funzione Row Fourier Transform:
 * calcola la trasformata di fourier sulle righe di
 * un array bidimensionale (in realta' un array
 * monodimensionale con righe*colonne elementi) */

complex<double>* rowFT(complex<double> *mat, complex<double> rootM){
  complex<double> * buff = new complex<double>[N*M];
  for (int i=0;i<N;i++){
    for(int l=0;l<M;l++){
      buff[M*i+l]=0;
      for(int k=0;k<M;k++){
        buff[M*i+l]=buff[M*i+l]+mat[M*i+k]*pow(rootM,(k*l)%M);
      }
    }
  }
  return buff;
}

complex<double>* aRowFT(complex<double> *mat, complex<double> rootM){
```

```
  complex<double> * buff = new complex<double>[N*M];
  for (int i=0;i<N;i++){
    for(int l=0;l<M;l++){
      buff[M*i+l]=0;
      for(int k=0;k<M;k++){
        buff[M*i+l]=buff[M*i+l]+mat[M*i+k]*pow(rootM,-((k*l)%M));
      }
      buff[M*i+l]=buff[M*i+l]/(double)(M);
    }
  }
  return buff;
}

/* funzione Column Fourier Transform:
 * calcola la trasformata di fourier sulle colonne di
 * un array bidimensionale (in realta' un array
 * monodimensionale con righe*colonne elementi) */

complex<double>* columnFT(complex<double> *mat,complex<double> rootN){
  complex<double> * buff = new complex<double>[N*M];
  for (int j=0;j<N;j++){
    for(int k=0;k<M;k++){
      buff[M*j+k]=0;
      for(int i=0;i<N;i++){
        buff[M*j+k]=buff[M*j+k]+mat[M*i+k]*pow(rootN,(i*j)%N);
      }
    }
  }
  return buff;
}

complex<double>* aColumnFT(complex<double> *mat,complex<double> rootN){
  complex<double> * buff = new complex<double>[N*M];
  for (int j=0;j<N;j++){
    for(int k=0;k<M;k++){
      buff[M*j+k]=0;
      for(int i=0;i<N;i++){
        buff[M*j+k]=buff[M*j+k]+mat[M*i+k]*pow(rootN,-((i*j)%N));
      }
      buff[M*j+k]=buff[M*j+k]/(double)(N);
    }
  }
  return buff;
}
```

```
complex<double>* creaA(){
  complex<double> * a = new complex<double>[N*M];
  for (int i=0;i<N; i++){
    for (int j=0;j<M;j++){
      if((i>=n) or (j>=m))
        a[i*M+j]=0;
      else
        a[i*M+j]=1;
    }
  }
  return a;
}

int main(){
  cout << "inserire p" << endl;
  cin >> p;
  while (!primo(p)){
    cout << "numero non primo, prego reinserire p" << endl;
    cin >> p;
  }
  cout << "inserire m" << endl;
  cin >> m;
  cout << "inserire n" << endl;
  cin >> n;
  while (!divide(n)){
    cout << "n non divide (p^m-1), prego reinserire n" << endl;
    cin >>n;
  }
  N = explog(2*n - 1);
  if ((p==2)or(divide(2*n)))
  M = explog(4*m -3);
  else{
  pm=true;
  }
  cout << "----inserire-polinomio-irriducibile ----"<<endl;
  double * irr = new double[M];
  double * xp = new double[m*(M-m+1)];
  for ( int t=0;t<M;t++){
    if(t<=m){
      cout << "inserire il coefficente numero "<<t<<endl;
      cin >> irr[t];
      if (irr[t]>=p){
        cout<<"coefficente non valido (maggiore di p)"<<endl;
```

```
        t--;
      }
    }
    else
      irr[t]=0;
}
calcoloX(irr, xp);
cout << "-----inserire-Alfa-----"<<endl;
double * alfa = new double[M];
double * provalfa = new double[M];
double * uno = new double[M];
bool alfaok = false;
bool fake;
for (int trd=1;trd<M;trd++)
  uno[trd]=0;
uno[0]=1;
while (!alfaok){
  fake=false;
  for ( int w=0;w<M;w++){
    if (w<m) {
      cout << "inserisci il coefficente numero "<<w<<endl;
      cin >> alfa[w];
      if (alfa[w]>=p){
        cout<<"coefficente non valido (maggiore di p)"<<endl;
        w--;
      }
    }
  else
  alfa[w]=0;
  }
  for (int st=0;st<M;st++)
  provalfa[st]=alfa[st];
  for (int rp=0;rp<n-1;rp++){
    provalfa = moltiplica(alfa, provalfa);
    if (grado(provalfa)>=m)
      modulo(provalfa, xp);
    else
      moduloP(provalfa);
    if ((uguali(provalfa,uno))and(rp!=n-2)){
      fake=true;
      rp=n-1;
    }
  }
  if (!(uguali(provalfa,uno))){
```

```
      cout << "non e' radice n-esima dell'unitÃ "<<endl;
    }
    else{
      if (fake==true){
        cout<< "e' radice n-esima non primitiva"<<endl;
      }
      else{
        alfaok=true;
      }
    }
  }
  cout<<"-----calcolo-Beta-----"<<endl;
  double * beta = new double[M];
  if (p==2){
  cout <<"p=2: esiste una radice di ordine 2n"<<endl;
    for (unsigned int l=0;l<m;l++)
      beta[l]=alfa[l];
    unsigned int sq=(n+1)/2;
    for (unsigned int s=0;s<sq-1;s++){
      beta = moltiplica(beta, alfa);
      if (grado(beta)>=m)
        modulo(beta, xp);
      }
    }
  else{
    if (!pm){
      cout << "2n|(p^m-1): esiste una radice di ordine 2n"<<endl;
      double * element= new double[M];
      double * buff =new double[M];
      for (int re=0;re<M;re++){
        element[re]=0;
        buff[re]=0;
      }
      bool found = false;
      int ii = 0;
      element[0]=2;
      while ((element[m-1]<p)and(found==false)){
        buff = moltiplica(element,element);
        if (grado(buff)>=m)
          modulo(buff, xp);
        else
          moduloP(buff);
        if (uguali(buff,alfa)){
          for(int iaa=0;iaa<m;iaa++)
```

```
            beta[iaa]=element[iaa];
          found = true;
        }
        element[0]=element[0]+1;
        while ((element[ii]==p)and(ii<m-1)){
          element[ii]=0;
          element[ii+1]=element[ii+1]+1;
          ii++;
        }
        ii=0;
      }
    }
    else{
      cout << "non esiste una radice 2n-esima in GF(p^m),"<<endl;
      cout << "sarebbe necessario immergersi in GF(p^2m)."<<endl;
      cout << "la cosa non e' prevista da questo algoritmo."<<endl;
    }
}
complex<double> rootN;
complex<double> rootM;
complex<double> * a = new complex<double>[N*M];
//"matrice" N*M contenente il vettore da trasformare
complex<double> * a_ = new complex<double>[N*M];
complex<double> * b = new complex<double>[N*M];
//"matrice" N*M contenente beta^i^2
complex<double> * b_ = new complex<double>[N*M];
complex<double> * y = new complex<double>[N*M];
//"matrice" N*M conetnente beta^-i^2
complex<double> * Y = new complex<double>[N*M];
complex<double> * x_ = new complex<double>[N*M];
complex<double> * X = new complex<double>[N*M];
complex<double> * z_ = new complex<double>[N*M];
complex<double> * Z = new complex<double>[N*M];
complex<double> * u = new complex<double>[N*M];
complex<double> * u_ = new complex<double>[N*M];
rootN = radice(N);
rootM = radice(M);
leggiMatrice(a);
cout << "----calcolo-della-trasformata----"<<endl;
double * beta_buff = new double[M];
double * beta_buff_inv = new double[M];
double * pettinati = new double[M];
beta_buff_inv = inverso(beta, xp);
for (int i_=0;i_<M;i_++){                //prepara il vettore b
```

```
    if (i_==0){                                    //come Beta alla i^2
      b[i_]=1;                                     //da 0 a n-1(deriva
      y[M*(n-1)+i_]=1;                             //dall'applicazione
    }                                              //di Bluestein) e il
    else{                                          //vettore y come Beta
      b[i_]=0;                                     //alla -i^2, da 1-n a n-1.
      y[M*(n-1)+i_]=0;
    }
    b[M+i_]=beta[i_];
    y[M*(n-2)+i_] = beta_buff_inv[i_];
    y[M*(n)+i_] = beta_buff_inv[i_];
  }
  stampaPolinomio(beta, grado(beta));
  for (int rw=2;rw<n;rw++){
    for (int i__=0;i__<M;i__++)
      beta_buff[i__]=beta[i__];
    for (int mol=0;mol<=(rw*rw)-2;mol++){
      beta_buff=moltiplica(beta_buff, beta);
      modulo(beta_buff, xp);
    }
    beta_buff_inv = inverso(beta_buff, xp);
    for (int j_=0;j_<M;j_++){
    b[M*rw+j_]=beta_buff[j_];
    y[M*(n-1-rw)+j_]=beta_buff_inv[j_];
    y[M*(n-1+rw)+j_]=beta_buff_inv[j_];
    }
  }
  //cout<<endl<<"-----a, b, y -----"<<endl;
  //stampaMatrice(a);
  //stampaMatrice(b);
  //stampaMatrice(y);
  //cout << "------a_, b_, x_------"<<endl;
  a_ = rowFT(a, rootM);                            //Step 1
  b_ = rowFT(b, rootM);
  for (int ik=0; ik<N*M; ik++)                     //step 2
    x_[ik]=a_[ik]*b_[ik];
  //stampaMatrice(a_);
  //stampaMatrice(b_);
  //stampaMatrice(x_);
  //cout << "-----X , Y , Z--------"<<endl;
  X = columnFT(x_, rootN);                         //Step 3
  Y = columnFT(rowFT(y, rootM), rootN);            //Step 4
  for (int iy=0; iy<N*M; iy++)                     //Step 5
    Z[iy]=X[iy]*Y[iy];
```

```
  //stampaMatrice(X);
  //stampaMatrice(Y);
  //stampaMatrice(Z);
  z_ = aColumnFT(Z, rootN);                    //Step 6
  for (int iN=n-1; iN<N; iN++){                //Step 7
    for (int iM=0;iM<M; iM++){
      u_[iN*M+iM]=z_[iN*M+iM]*b_[(iN-n+1)*M+iM];
    }
  }
  u = aRowFT(u_, rootM);                       //Step 8
  cout << endl<<"-----risultato-----"<<endl;
//stampaMatrice(u);
  double * risulta_pol = new double[M];
  double * risulta = new double[m*n];
  for (int llj=n-1; llj<2*n-1; llj++){
    for (int lli=0; lli<M; lli++){
      risulta_pol[lli]=real(u[llj*M+lli]);}
    if (grado(risulta_pol)>=m)
      modulo(risulta_pol,xp);
    else
      moduloP(risulta_pol);
    for (int llk=0;llk<m; llk++){
      if (risulta_pol[llk]<0.5)
        risulta_pol[llk]=0;
      if ((risulta_pol[llk]>p-1)and(risulta_pol[llk]<p+1))
        risulta_pol[llk]=0;
      risulta[(llj-n+1)*m+llk]=risulta_pol[llk];
      }
    cout << endl;
  }
  stampaMatrice(risulta);
}
```

Bibliografia

[1] Leo I. Bluestein. A linear filtering approach to the computation of discrete Fourier transform. *Audio and Electroacoustics, IEEE Transactions on*, 18(4):451–455, 1970.

[2] Cormen, T. H.Leiserson, C. E.Rivest, R. L.Stein, C. . *Introduction to Algorithms.* The MIT Press, 1990.

[3] N. Lauritzen. *Concrete Abstract Algebra.* Cambridge University Press, 2003.

[4] Rudolf Lidl and Harald Niederreiter. *Finite fields.* Cambridge University Press, 1997.

[5] J. M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–365, May 1971.

[6] Franco P. Preparata and Dilip V. Sarwate. Computational complexity of Fourier transforms over finite fields. *Mathematics of Computation*, 31(139):740–740, September 1977.

[7] Ezio Stagnaro. Algebra Commutativa o Geometria Algebrica Affine.