



UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di Laurea Magistrale in Informatica

Metodi del Calcolo Scientifico - Progetto 2

Compressione di immagini tramite la DCT

Alberici Federico - 808058

Bettini Ivo Junior - 806878

Cocca Umberto - 807191

Traversa Silvia - 816435

Anno Accademico 2019 - 2020

Indice

Introduzione	2
Analisi DCT	3
1.1 Discrete Cosine Transform	3
1.2 DCT e IDCT	3
1.3 DCT2 e IDCT2	5
1.4 Varianti DCT	7
1.5 Confronto	8
Test con immagini	9
2.1 Tool di testing	9
2.2 Risultati	10

Introduzione

In questa relazione vengono presentate e discusse le modalità di implementazione della DCT (dall'inglese Discrete Cosine Transform), ovvero la più diffusa funzione che provvede alla compressione spaziale.

Nella prima parte viene confrontata la versione nativa delle DCT implementata in questo progetto con alcune varianti fast (FFT), studiandone il costo computazionale.

Nella seconda parte viene documentato un semplice tool per applicare su immagini in toni di grigio, tramite un approccio di compressione di tipo jpeg (senza utilizzare una matrice di quantizzazione), la funzione DCT2 implementata.

Analisi DCT

1.1 Discrete Cosine Transform

Una DCT esprime una sequenza finita di punti in termini di una somma di funzioni coseno oscillanti a diverse frequenze. Ad oggi è una delle tecniche di trasformazione più utilizzate nella Teoria dei segnali e nella compressione dei dati, in particolare nei media digitali (audio, video, radio ecc..).

In queste applicazioni infatti la maggior parte delle informazioni significative tendono ad essere concentrate in poche componenti a bassa frequenza della DCT. Questo permette di comprimere a piacere il dato scartando le componenti ad alta frequenza (compressione lossy).

1.2 DCT e IDCT

La DCT-II è probabilmente la forma più utilizzata, infatti viene indicata come "la DCT".

$$C_k = \alpha_k \sum_{i=0}^{N-1} V_i \cos \left[\frac{\pi (2i+1) k}{2N} \right] \quad i = 0, \dots, N-1 \quad e \quad \alpha_k = \begin{cases} 1/\sqrt{N}, & \text{se } k = 0 \\ \sqrt{2/N}, & \text{se } 1 \leq k \leq N-1 \end{cases}$$

La sua inversa è la DCT-III e per questo viene indicata come "l'inversa della DCT" o "IDCT".

$$V_i = \sum_{k=0}^{N-1} \alpha_k C_k \cos \left[\frac{\pi (2i+1) k}{2N} \right] \quad k = 0, \dots, N-1 \quad e \quad \alpha_k = \begin{cases} 1/\sqrt{N}, & \text{se } k = 0 \\ \sqrt{2/N}, & \text{se } 1 \leq k \leq N-1 \end{cases}$$

Entrambe le funzioni effettuano N somme per calcolare la k-esima componente di un vettore di N componenti, determinando un costo computazionale $O(N^2)$.

Implementazione

Per l'implementazione è stato utilizzato C++, sfruttando la libreria open-source Eigen (<https://eigen.tuxfamily.org/>) per semplificare la gestione dei dati.

```

1  void DCT2::DCT(Eigen::VectorXd &_v)
2  {
3
4      const Eigen::VectorXd _v_copy = _v;
5      const int N = _v.size();
6      double ak = 1.0 / sqrt(N);
7      double ck = 0;
8
9      for (int k = 0; k < N; k++)
10     {
11         ck = 0;
12         for (int i = 0; i < N; i++)
13         {
14             ck += cos((2.0 * i + 1.0) * k * M_PI / (2.0 * N)) * _v_copy(i);
15         }
16         _v(k) = ak * ck;
17         if (k == 0)
18         {
19             ak = sqrt(2.0) / sqrt(N);
20         }
21     }
22 }
```

Listato 1.1: Funzione di calcolo DCT

```

1 void DCT2::IDCT(Eigen::VectorXd &_c)
2 {
3     const Eigen::VectorXd _c_copy = _c;
4     const int N = _c.size();
5     double ak = 0;
6     double vi = 0;
7
8     for (int i = 0; i < N; i++)
9     {
10         vi = 0;
11         ak = 1.0 / sqrt(N);
12         for (int k = 0; k < N; k++)
13         {
14             vi += cos((2.0 * i + 1.0) * k * M_PI / (2.0 * N)) * _c_copy(k) * ak;
15             if (k == 0)
16             {
17                 ak = sqrt(2.0) / sqrt(N);
18             }
19         }
20         _c(i) = vi;
21     }
22 }

```

Listato 1.2: Funzione di calcolo IDCT

1.3 DCT2 e IDCT2

La DCT2 è una trasformazione a due dimensioni, ottenuta semplicemente applicando la DCT mono-dimensionale ad una dimensione, seguita da un'altra applicazione all'altra dimensione.

La definizione della DCT bi-dimensionale per una matrice A $m \times n$ in input è:

$$C_{kl} = \alpha_k \alpha_l \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} A_{ij} \cos \left[\frac{\pi (2i+1) k}{2m} \right] \cos \left[\frac{\pi (2j+1) l}{2n} \right],$$

$$\text{con } 0 \leq k \leq m-1, \quad 0 \leq l \leq n-1,$$

$$\alpha_k = \begin{cases} 1/\sqrt{m}, & \text{se } i = 0 \\ \sqrt{2/m}, & \text{se } 1 \leq i \leq m-1 \end{cases} \quad e \quad \alpha_l = \begin{cases} 1/\sqrt{n}, & \text{se } j = 0 \\ \sqrt{2/n}, & \text{se } 1 \leq j \leq n-1 \end{cases}$$

L'inversa di tale trasformazione è la IDCT2, ottenuta applicando IDCT alle due dimensioni.

Implementazione

Anche in questo caso Eigen è utilizzato per mantenere la struttura dati tramite un oggetto *Eigen::MatrixXd*. L'implementazione non sfrutta direttamente la definizione ma computa la DCT2/IDCT2 prima sulle righe e poi sulle colonne della matrice in input. Inoltre essendo l'elaborazione di ogni vettore indipendente dagli altri è possibile parallelizzarne la computazione.

```
1 Eigen::MatrixXd DCT2::DCT2_mt(Eigen::MatrixXd &m)
2 {
3     Eigen::MatrixXd out = m;
4
5     // DCT su righe
6     #pragma omp parallel for
7     for (int i = 0; i < out.rows(); i++)
8     {
9         Eigen::VectorXd row = out.row(i);
10        DCT(row);
11        out.row(i) = row;
12    }
13
14    // DCT su colonne
15    #pragma omp parallel for
16    for (int i = 0; i < out.cols(); i++)
17    {
18        Eigen::VectorXd col = out.col(i);
19        DCT(col);
20        out.col(i) = col;
21    }
22
23    return out;
24 }
```

Listato 1.3: Funzione di calcolo DCT2

```

1  Eigen::MatrixXd DCT2::IDCT2_mt(Eigen::MatrixXd &m)
2  {
3      Eigen::MatrixXd out = m;
4
5      // IDCT su righe
6      #pragma omp parallel for
7          for (int i = 0; i < out.rows(); i++)
8          {
9              Eigen::VectorXd row = out.row(i);
10             IDCT(row);
11             out.row(i) = row;
12         }
13
14     // IDCT su colonne
15     #pragma omp parallel for
16         for (int i = 0; i < out.cols(); i++)
17         {
18             Eigen::VectorXd col = out.col(i);
19             IDCT(col);
20             out.col(i) = col;
21         }
22
23     return out;
24 }

```

Listato 1.4: Funzione di calcolo IDCT2

1.4 Varianti DCT

Esistono diverse varianti della DCT che riducono la complessità a $O(N \log N)$. Tali metodi sono conosciuti come *fast DCT* o *FCT* in quanto appunto migliorano notevolmente il costo computazionale.

Di seguito vengono citate due delle più comuni.

Fast DCT di Lee

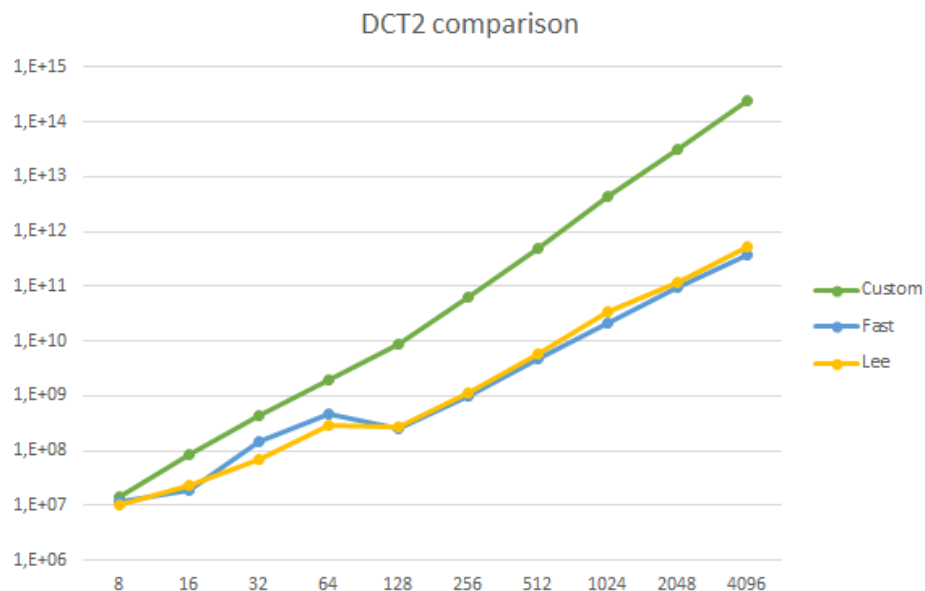
Descritta da Byeong Gi Lee [Lee] nel 1984 è uno degli algoritmi fast DCT per 2^m punti più comune. Utilizza una struttura ricorsiva dove la trasformazione DCT è decomposta in una parte pari e una dispari. Queste parti sono a loro volta decomposte nello stesso modo finché non sono abbastanza piccole ($N=2$) da essere calcolate tramite valutazione

diretta [LAGERSTRM2001DesignAI].

Fast DCT FFT

Invece di applicare direttamente la formula DCT (o scomporla come mostrato da Lee) è possibile fattorizzare la computazione in modo simile alla *fast Fourier transform* (FFT). Gli algoritmi basati su sull'algoritmo di Cooley-Tukey [10.2307/2003354] sono i più comuni, ma qualunque altro algoritmo FFT è applicabile.

1.5 Confronto



Test con immagini

Il programma è stato scritto tramite QT, una libreria multiplatforma per lo sviluppo di programmi che utilizzano un'interfaccia grafica (attraverso l'uso di widget) che utilizza il linguaggio C++ (motivo per il quale abbiamo deciso di utilizzarlo).

Nella seconda parte del progetto, il programma utilizza le seguenti classi scritte da noi:

- **main.cpp**, la quale si occupa di eseguire l'intero corpo del programma;
- **compress.cpp**, la quale data un'immagine x esegue delle iterazioni per creare delle sottomatrici che verranno utilizzate poi dalla libreria `dct2` (presa da noi online, da sistemare). Inoltre, una volta che la libreria `dct2` esegue il calcolo, questa classe si occupa di ricomporre l'immagine finale;
- **mainwindow.cpp**, classe che gestisce tutti gli aspetti e i trigger della interfaccia grafica;

Inoltre, viene utilizzata la libreria `DCTFast` nella quale ricaviamo la `DCT2` operando prima per righe e poi per colonne.

2.1 Tool di testing

Una volta avviato il programma, è possibile caricare l'immagine `.bmp` tramite un apposito tasto ed inserire i valori di f e d . Una volta inseriti i parametri, tramite il pulsante `process` viene chiamato il metodo `on_parameters_clicked()`, che dopo aver controllato che f e d rispettino tutti i vincoli trasforma l'immagine (tramite la funzione `pixmapToMatrix()`) in una matrice e la invia alla funzione `DCTcompress`. Questa funzione divide l'immagine

in blocchi, applica la DCT2 e poi restituisce una matrice che viene passata alla funzione `matrixToPixmap()` per poter essere visualizzata in output nell'interfaccia grafica.

La funzione `compress` prende in input una matrice di interi e i parametri `f` e `d` restituendo in output una matrice di interi. Al suo interno, la matrice di input viene trasformata nel formato `double` e viene eseguito un troncamento per poter scartare gli "avanzi". Iterativamente, per ogni blocco quadrato $f \times f$ applichiamo la `dct2` e poi ritorniamo la matrice nel formato `int` (arrotondando o suoi valori `double` all'intero più vicino, mettendo a zero i valori negativi e a 255 quelli maggiori di 255).

2.2 Risultati