FIC: Fractal Image Compression

L. De Sano, A. Donizetti

Settembre 2015

Indice

1 Introduzione

Con il termine "Fractal image Compression" si va ad indicare una famiglia di tecniche di compressione di immagini (o video) basate sulle proprietà matematiche dei frattali. Tali metodi di compressione si rivelano sopra ogni altra cosa adatti a comprimere textures e immagini naturali, o, più in generale, immagini che sono caratterizzate da un elevato livello di *self-similarity* (ovvero aventi delle parti che, al netto di rotazioni e ingrandimenti/riduzioni, somigliano ad altre parti dell'immagine).

La compressione di immagini tramite frattali (così come altre, più diffuse, ad esempio JPEG) appartiene a quel gruppo di tecniche di compressione *lossy*, ovvero in cui la compressione dell'immagine avviene al costo di una perdita di informazione. Tuttavia, a differenza di quanto accade quando si utilizza uno dei metodi di compressione basati sui pixel (come JPEG, GIF o MPEG), nella compressione frattale nessuna parte dell'immagine viene effettivamente memorizzata. Ciò che viene memorizzato è invece la *struttura interna* dell'immagine (ad esempio un indice di quali parti, effettuate le dovute trasformazioni, sono simili ad altre parti).

Poiché nessun pixel dell'immagine originale viene memorizzato, la decompressione parte da un singolo pixel, di colore qualsiasi, e procede alla ricostruzione dell'immagine originale applicando iterativamente una mappa ricavata dalla struttura interna dell'immagine originale.

In questo documento tratteremo delle tecniche di compressione di immagini basate su frattali, iniziando con una panoramica teorica del loro funzionamento, proseguendo con la discussione di alcuni aspetti pratici e presentando una implementazione giocattolo realizzata in MATLAB, e concludendo infine con alcuni test che consentiranno di valutare praticità e *performances* di una libreria di compressione basata su frattali, anche in confronto con altre tecniche di compressione *lossy* maggiormente utilizzate.

2 Aspetti teorici

2.1 Introduzione ai frattali

Un frattale è un oggetto geometrico che si ripete nella sua forma, allo stesso modo, su diverse scale. Questo comportamento fa si che ingrandendo una sua qualsiasi componente, ciò che si ottiene è una figura simile all'originale. In linea di massima, si può dire che perché l'insieme F sia considerato un frattale, F dovrebbe avere almeno le seguenti proprietà:

- F ha dettagli ad ogni scala d'ingrandimento;
- F gode di autosimilitudine (a qualunque scala si osservi, presenta sempre le stesse caratteristiche globali);
- la dimensione frattale[1] di F è maggiore della sua dimensione topologica[2];
- esiste un algoritmo relativamente semplice per costruire F.

Un esempio di oggetto geometrico secondo i principi di costruzione di un frattale e che rispetta le proprietà elencate è la curva di Koch. La costruzione comincia con una linea di lunghezza 1 chiamata *initiatior*. Da questa linea si rimuove il terzo centrale e lo si sostituisce con due linee della stessa lunghezza della parte rimossa. Questa nuova forma viene chiamata *generator*. La prima parte della costruzione è mostrata in figura 2.1.

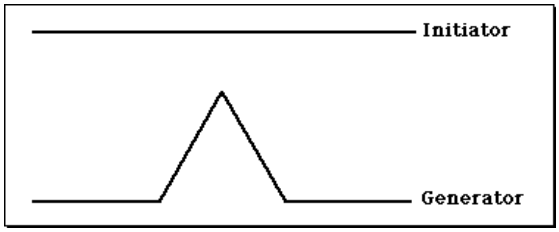

Figura 2.1: Initiator e generator per la curva di Koch

La regola può essere nuovamente applicata su ogni linea, così da andare a sostituirla ogni volta come fanno nel passaggio da *initiator* a *generator*. Il secondo livello è visibile in figura 2.2.

[1] La dimensione frattale è un rapporto che fornisce un indice statistico relativo a come varia la complessità di un frattale rispetto alla scala a cui viene misurato.

[2] La dimensione topologica è un concetto di dimensione che si applica ad uno spazio topologico, ad

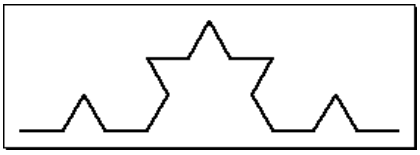

Figura 2.2: Livello 2 per la curva di Koch

Una volta che la procedura è avviata può proseguire a piacimento. Il terzo e il quarto livello sono visibili nelle figure 2.3 e 2.4 rispettivamente.

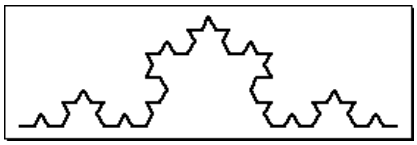

Figura 2.3: Livello 3 per la curva di Koch

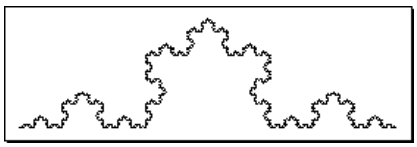

Figura 2.4: Livello 4 per la curva di Koch

2.2 Iterated Function Systems

Uno strumento matematico che si rivelerà fondamentale nella fase di decompressione delle immagini trattate con un metodo di compressione frattale è quello degli *Iterated Function Systems*, che andiamo per questo motivo a descrivere qui.

Senza eccedere in formalità, definiamo un *Iterated Function System* (IFS) come una collezione di trasformazioni contrattive $\{w_i : \mathbb{R}^2 \to \mathbb{R}^2\}_{i=1,\dots,n}$ che mappa il piano su se stesso. Questa collezione di trasformazioni definisce una mappa

$$W(\cdot) = \bigcup_{i=1}^{n} w_i(\cdot)$$

Tale mappa è applicata ad insiemi di punti, ed il risultato è definito in questa maniera: dato un insieme di punti $S \in \mathbb{R}^2$, calcoliamo $w_i(S)$ per ogni i (ovvero produciamo i copie ridotte di S), e poi calcoliamo l'unione dei risultati (ovvero, mettiamo insieme tutte le copie ridotte), ottenendo così un nuovo insieme $S' = W(S)$.

esempio in $\mathbb{R}^n$ è n.

W così definito è una mappa tra sottoinsiemi di $\mathbb{R}^2$ (che d'ora in avanti per noi saranno "immagini"), e gode di due importanti (e rilevanti per la compressione frattale) proprietà, che andiamo ad enunciare senza dimostrazione.

Proprietà 1 *Se tutte le w_i sono contrattive, allora W è contrattiva in uno spazio di sottoinsiemi del piano.*

Proprietà 2 *Fissata una mappa W, esiste una immagine x_W, chiamata* attrattore per *W, tale che*

- $W(x_W) = x_W$
- *data una qualunque immagine di partenza S_0, vale che*

$$x_W \equiv \lim_{n\to\infty} W^n(S_0)$$

  ovvero l'applicazione ripetuta per n volte di W porta ad ottenere x_W, per n sufficientemente grande, ed indipendentemente dalla scelta dell'immagine di partenza.

- *x_W è unica. Se una qualsiasi immagine S soddisfa $W(S) = S$, allora S è l'attrattore per W.*

La seconda di queste proprietà è nota come *Contractive Mapping Fixed-Point Theorem.*

2.3 Self-similarity nelle immagini

Immagini come oggetti matematici

Per poter applicare la teoria degli *IFS* alle immagini come comunemente le intendiamo (ovvero, in sostanza, matrici di pixels rappresentati da 1 byte – *greyscale* – oppure più di uno – es. RBG –), è necessario formalizzare in qualche modo il concetto di "immagine digitale", cercando di inserirlo in un contesto matematico che ci consenta di manipolarla utilizzando di strumenti messi a disposizione dall'algebra. Per semplicità, ci limiteremo qui a considerare le immagini in *grayscale* (un singolo canale di colore, l'intensità indicata con un numero da 0 a 255).

Consideriamo un'immagine in scala di grigi, memorizzata come matrice di pixel, 1 byte per ogni pixel. Non è difficile immaginare di poter rappresentare tale immagine utilizzando una funzione $f : \mathbb{R}^2 \to \{1, 2, \ldots, 255\}$, in modo tale che ad ogni pixel alle coordinate (x, y) sia fatto corrispondere un punto (x, y) nel piano, al quale a sua volta viene associata un'altezza z, che corrisponde al valore di grigio del pixel in questione.

Ad esempio, ad un'immagine con il primo pixel in basso a sinistra avente livello di grigio 177, viene fatta corrispondere una funzione f che, per quanto riguarda quel preciso punto, avrà $f(0, 0) = 177$. Effettuando questa operazione su tutti i pixel dell'immagine originale, è possibile definire completamente f su tutta la parte del piano di nostro interesse (e, in genere, si riduce per convenzione l'immagine ad avere dominio $[0, 1]^2$). Nella

figura qui sotto riportata si può vedere un esempio del risultato finale della procedura, applicata ad una immagine grayscale 128×128 pixels.

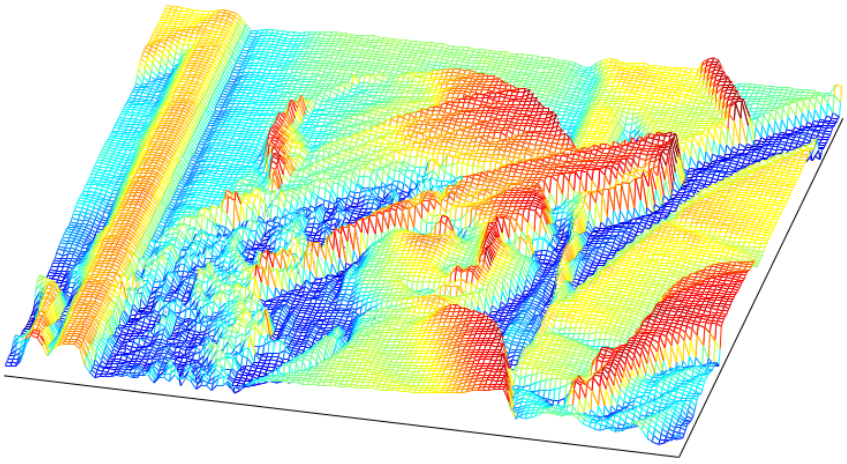

Figura 2.5: lena grayscale, 128×128 pixels, trasformata in funzione $[0,1]^2 \to \{0, \ldots, 255\}$

La trasformazione descritta ci permette di ignorare l'aspetto "visivo" della modalità in cui una immagine è memorizzata, e ci consente di identificare una immagine con la funzione f associata, su cui si andrà ad operare durante il procedimento di compressione.

Distanza tra due immagini

Un secondo concetto che è necessario introdurre per poter operare sulle immagini durante il procedimento di compressione e decompressione è quello di "distanza" tra due immagini. La contrattività di una mappa W (ovvero, intuitivamente, il fatto che essa rende le immagini più "piccole") può essere verificata solo se si ha a disposizione una qualche metrica sulla base della quale è possibile valutare la distanza tra due sottoinsiemi del piano (ovvero, nel nostro caso, due immagini).

Date due immagini f, g, definiamo la loro distanza $d(f, g)$, utilizzando la media quadratica, come

$$d(f,s) = \sqrt{\int_{[0,1]^2} f(x,y) - g(x,y) \quad \mathrm{d}x\mathrm{d}y}$$

La metrica ci fornisce una maniera di valutare la distanza tra due immagini pesando in maniera uniforme tutti i punti in esse contenuti.

Self-similarity debole

A differenza di quanto accade quando si osserva un frattale vero e proprio, all'interno di immagini "naturali" non troviamo una *self-similarity* forte: solitamente non abbiamo parti dell'immagine che sono simili all'immagine complessiva, bensì parti dell'immagine che sono simili ad altre parti dell'immagine. Un altro aspetto da considerare è quello della colorazione: due parti dell'immagine potrebbero essere simili nella composizione dei pixel, ma differenti per quanto riguarda la gradazione di grigio dei pixels che le compongono.

I basilari IFS che abbiamo descritto in precedenza vanno leggermente complicati, nel contesto della compressione frattale delle immagini, in modo da accomodare i due aspetti qui sopra descritti. Come prima cosa, notiamo che al momento di definire le trasformazioni w_i, dovremo fare in modo che ad ogni trasformazione sia consentito di andare a toccare solo un particolare sottoinsieme del piano, in maniera da accomodare il requisito di poter rappresentare la *self-similarity* debole (similarità tra parti e altre parti, e non con l'intera immagine). Come seconda cosa, alle mappe vanno aggiunti due parametri, *contrasto* e *luminosità*, che ci consentiranno di definire non solo trasformazioni *spaziali*, ma anche trasformazioni che vanno a coinvolgere la differenza di *colorazione* (sempre su scala di grigi) tra due sottoinsiemi di piano.

Invece della classica trasformazione su $\mathbb{R}^2$ che ci aspetteremmo considerato il fatto che stiamo operando su funzioni $\mathbb{R}^2 \to \mathbb{R}$, definiamo le w_i come trasformazioni su $\mathbb{R}^3$:

$$w_i \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix}$$

La trasformazione è formata da una componente *spaziale*, ovvero

$$v_i \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix}$$

e di due parametri aggiuntivi s_i ed o_i, che agiscono rispettivamente come moltiplicatore e scala tramite somma sulla componente z della funzione. Questi due parametri, che chiamiamo rispettivamente *contrasto* e *luminosità*, non influenzano in alcun modo le componenti spaziali della trasformazione, e ci consentono di definire trasformazioni sullo spazio di *colore* dell'immagine (che, ricordiamo, è rappresentato nella funzione associata all'immagine come la componente z).

Ricordando che l'immagine è rappresentata da una funzione $z = f(x, y)$, dove x ed y sono pixels e z il grado di grigio dell'immagine *grayscale*, una singola trasformazione w_i viene applicata calcolando $w_i(f) = w_i(x, y, f(x, y))$. La parte spaziale di w_i determina come una sotto-parte dell'immagine è mappata su altre sottoparti, mentre s_i ed o_i determinano contrasto e luminosità della trasformazione.

Il fatto che il mapping sia da effettuarsi da sottoinsiemi a sottoinsiemi dell'immagine è permesso dal fatto che le varie w_i sono implicitamente definite in modo da operare su un sottoinsieme di $[0,1]^2$, che chiamiamo D_i, dando come risultato un sottoinsieme di $[0,1]^2$, che chiameremo R_i. In simboli, scriviamo che

$$v_i(D_i) = R_i$$

ovvero la componente spaziale della trasformazione ha dominio in sottoinsiemi del piano che chiamiamo D_i e codominio in sottoinsiemi del piano che chiamiamo R_i.

Poiché richiediamo che $W(f)$, l'unione delle w_i, sia a sua volta un'immagine compatta, le trasformazioni devono soddisfare due proprietà basilari: prima di tutto gli R_i devono partizionare completamente $[0,1]^2$, ovvero deve valere $\bigcup_i R_i = [0,1]^2$. Inoltre gli R_i non devono sovrapporsi, ovvero deve valere $R_i \neq R_j$ se $i \neq j$. Queste due proprietà garantiscono che l'applicazione della mappa W su una immagine S risulti in una immagine S', possibilmente differente, ma completa ed univoca.

Poiché la mappa W è contrattiva solo se lo sono le trasformazioni w_i, è necessario scegliere accuratamente i parametri di queste ultime. Dato che la metrica di distanza d che abbiamo definito sopra non prende in considerazione le direzioni x ed y, ma soltanto la direzione z, per assicurare la contrattività delle w_i è sufficiente fare attenzione al parametro di scala s_i. In particolare:

Proprietà 3 *Le trasformazioni w_i sono contrattive per $s_i < 1$*

Decoding

A questo punto abbiamo tutti gli strumenti che servono per effettuare il *decoding* di una immagine compressa utilizzando frattali: partendo da una immagine qualunque, si applica ripetutamente la mappa W fino a quando non si raggiunge il suo punto fisso x_W, che rappresenta il risultato dell'operazione di decodifica.

L'aspetto interessante (e complicato) della tecnica di compressione che stiamo descrivendo è però un altro: come fare, data una immagine f da comprimere, a trovare un insieme di mappe $\{w_i\}_i$ tale che f sia il punto fisso della loro unione (W)? Questa operazione, che rappresenta il nucleo delle tecniche di compressione tramite frattali, è l'argomento della prossima sezione.

2.4 Encoding

Come anticipato nella precedente sezione, per comprimere un'immagine f è necessario trovare una mappa W tale che

$$f = W(f)$$

ovvero una mappa W avente f come attrattore. Ricordando la definizione di W, la proprietà richiesta diventa

$$f = w_1(f) \cup w_2(f) \cup \cdots \cup w_N(f)$$

Per ottenere questo, dovremmo procedere con il partizionare l'immagine f in tanti pezzi tali per cui, applicando le trasformazioni $\{w_i\}_i$, si otterrebbe l'immagine di partenza. Chiaramente, in generale, questa è una condizione troppo restrittiva: ottenere *esattamente* l'immagine di partenza non è qualcosa che sia possibile sperare (a parte nei casi in cui si stia cercando di comprimere un'immagine frattale!).

Quello che facciamo in pratica è cercare delle trasformazioni che, applicate, ci restituiscano un'immagine *differente*, diciamo f', tale per cui la distanza con l'immagine di partenza, ovvero $d(f, f')$, è piccola. Operativamente, procediamo calcolando le w_i con l'obbiettivo di minimizzare la distanza tra il pezzo di immagine che stiamo attualmente considerando e il pezzo di immagine ottenuto dopo l'applicazione della trasformazione. In formula, minimizziamo

$$d\big(f \cap (R_i \times I), w_i(f)\big)$$

per tutti gli i. In termini di R_i ed D_i (i concreti pezzi di immagini su cui andremo ad operare), dobbiamo partizionare l'immagine f in una collezione di pezzi R_i, e cercare da una seconda collezione D_i dei pezzi tali per cui la mappa da D_i ad R_i comporta un errore (calcolato come distanza d) che sia piccolo.

Metrica

Come abbiamo visto, la comparazione tra le due immagini avviene tramite la radice della media quadratica. L'utilizzo di questa metrica ci consente di calcolare i valori ottimali per s_i e di o_i in maniera diretta. In particolare, dati due quadrati contenenti n valori di luminosità dei pixel $a_1, \ldots, a_n$ per D_i e $b_1, \ldots, b_n$ per R_i, possiamo cercare i valori di s e di o che minimizzano la quantità:

$$R = \sum_{i=1}^{n} (s \cdot a_i + o - b_i)$$

Questa quantità ci fornisce i valori di contrasto e luminosità che fanno si che la trasformazione dai i valori di a_i a quelli di b_i abbia la minima distanza possibile. Il valore minimo di R si ha quando le derivate parziali di s e di o si azzerano, ovvero quando

$$s = \frac{n\sum_{i=1}^{n} a_i b_i - \sum_{i=1}^{n} a_i \sum_{i=1}^{n} b_i}{n\sum_{i=1}^{n} a_i^2 - \left(\sum_{i=1}^{n} a_i\right)^2}$$

e

$$o = \frac{1}{n}\left(\sum_{i=1}^{n} b_i - s\sum_{i=1}^{n} a_i\right)$$

A questo punto, avendo i valori ottimali di s e di o, possiamo calcolare R come:

$$R = \frac{1}{n}\left[\sum_{i=1}^{n} b_i^2 + s\left(s\sum_{i=1}^{n} a_i^2 - 2\sum_{i=1}^{n} a_i b_i + 2o\sum_{i=1}^{n} a_i\right) + o\left(no - 2\sum_{i=1}^{n} b_i\right)\right]$$

la distanza tra le due immagini A e B è ora equivalente a $\sqrt{R}$.

Partizionare le immagini

Una questione che è necessario innanzitutto risolvere è quella relativa alla scelta delle collezioni R_i e D_i.

La maniera più ovvia di partizionare f è quella di dividerla in quadrati di una dimensione predeterminata, ma questa scelta presenta dei problemi. Nel caso all'interno dell'immagine da trattare non vi sia una uniforme distribuzione della "complessità dei dettagli" (succede, ad esempio, per un ritratto di persona, che avrà un alto livello di complessità nella zona dove si trova il viso del soggetto e un basso livello di complessità ai bordi, dove si vede un muro di colore uniforme che fa da sfondo alla fotografia) una divisione uniforme andrà a penalizzare le zone con maggior presenza di dettagli, e a sprecare inutilmente blocchi (che saranno tutti uguali) per lo sfondo.

Un metodo di partizionamento migliore è quello basato sui *quadtrees*. Si divide inizialmente l'immagine in 4 parti, e si controlla il livello di fedeltà dei sotto-quadrati rispetto all'immagine orignale. Se il livello di fedeltà è entro un limite e_c prefissato, il blocco entra a far parte dell'insieme degli R_i. Altrimenti lo si partiziona in 4 parti e si ri-applica nuovamente, ricorsivamente, il procedimento appena descritto.

In genere è anche conveniente, nel contesto della compressione di immagini, scegliere due parametri $Q_{\max}$ e $Q_{\min}$ che definiscono rispettivamente la grandezza massima e minima dei blocchi R_i. Nel caso un blocco abbia dimensioni superiori a $Q_{\max}$, lo si partiziona a prescindere dalla complessità interna; nel caso un blocco avente dimensioni $Q_{\min}$ si cessa di partizionarlo, a prescindere dal livello di uniformità interna. Questi due parametri

consentono di avere un maggior controllo sulle dimensioni dei blocchi R_i, e soprattutto sulla cardinalità dell'insieme degli R_i, che influenza pesantemente il tempo di esecuzione dell'operazione di codifica.

L'immagine sottostante fornisce un esempio del tipo di partizione che si può ottenere utilizzando un *quadtree*. Si nota che nei punti in cui l'immagine è uniforme i blocchi sono di dimensioni maggiori, mentre nei punti ricchi di dettagli (il viso, la piuma del cappello) la suddivisione è tanto fine quanto lo consente il parametro $Q_{\min}$.

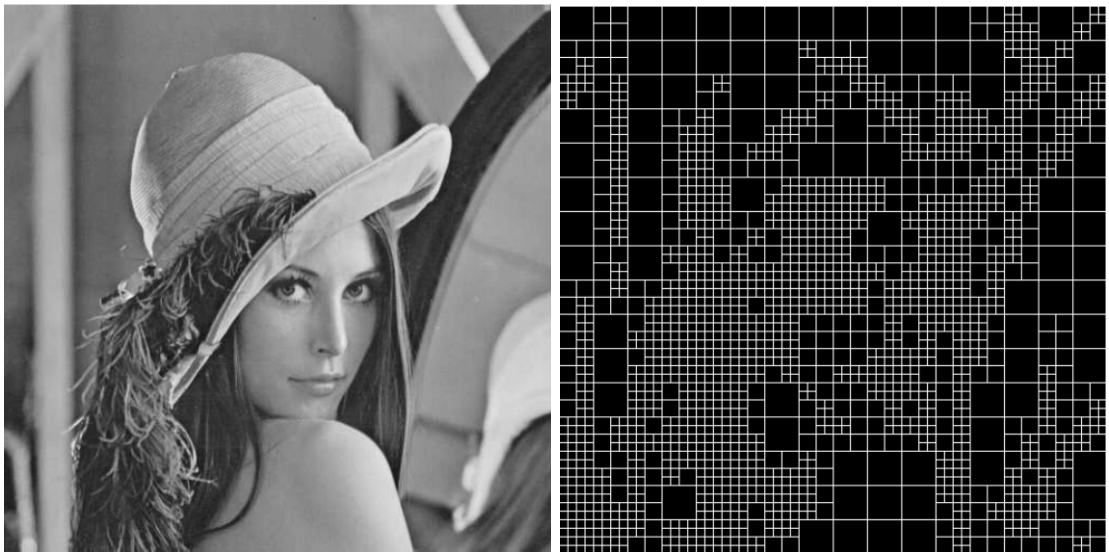

Figura 2.6: Quadtree su Lena, $Q_{\min} = 8$, $Q_{\max} = 64$, $e_c = 0.4$

Oltre al partizionamento con *quadtree*, esistono altri metodi applicabili in questo contesto, ad esempio il metodo di partizionamento HV ed il metodo di partizionamento triangolare, che va a generare sottoimmagini non quadrate e può essere quindi vantaggioso nei casi in cui l'immagine di partenza contenga un gran numero di linee oblique (che non sono facilmente coperte da metodi di partizionamento basati su quadrati).

L'algoritmo

Abbiamo ora a disposizione tutti gli elementi che ci servono per descrivere ad alto livello il funzionamento dell'algoritmo di codifica frattale. Lo riportiamo di seguito:

procedure ENCODE(e_c, $r_{\min}$)
    $R_1 \leftarrow [0,1]^2$
    $R \leftarrow R \cup R_1$
    Segna R_1 come non coperto
    while $\exists R_i \in R$ non coperto **do**

```
        D_i, w_i ← BESTCOVER(R_i)
        if d(f ∩ (R_i × I), w_i(f)) < e_c OR size(R_i) ≤ r_min then
            Segna R_i come coperto
            W ← W ∪ w_i
        else
            Partiziona R_i in 4 parti R_a, R_b R_c, R_d
            R ← R \ R_i
            R ← R ∪ {R_a, R_b, R_c, R_d}
            Segna R_{a,b,c,d} come non coperti
        end if
    end while
end procedure
```

La procededura BESTCOVER prende come input una sottimmagine R_i, e ritorna la sottoimmagine D_i e la rispettiva trasformazione w_i che coprono meglio R_i.

```
procedure BESTCOVER(R_i)
    score ← ∞
    d, w ←, NULL, NULL
    for all D_i ∈ D do
        w_temp, s ← SCORE(D_i, R_i)
        if s < score then
            w ← w_temp
            score ← s
            d ← D_i
        end if
    end for
    return d, w
end procedure
```

La procedura SCORE restituisce una trasformazione w_i ed uno *score* che indica il livello di somiglianza tra la sottoimmagine di partenza R_i e la sottoimmagine calcolata D_i.

3 Aspetti Pratici e implementazione

In questo capitolo forniamo un'implementazione didattica dell'algoritmo di compressione usando frattali per permettere di comprenderne gli aspetti pratici . Il codice è stato scritto in linguaggio MATLAB ed è disponibile all'indirizzo `github.com/luca-dex/shiny-octo-dubstep`.

3.1 Creazione del Dominio

Il primo passo dell'algoritmo è quello di andare a creare, partendo dall'immagine da comprimere, il dominio D. Questa operazione è descritta nel frammento di codice riportato di seguito.

```
function [ d, d2 ] = domains( image, range_sizes, l )
if nargin < 3
    l = 2;
end
d = {1000,2};
dim = length(image);
index = 1;
counter = 1;
range_sizes = sort(range_sizes, 'descend');
for size = range_sizes
    step = size / l;
    for i = 1:step:(dim-size)
        for j = 1:step:(dim-size)
            counter = counter + 1;
            imm = image(i:i+size-1, j:j+size-1);
            if check_equals(d, imm, index-1)
                continue
            end
            d(index,1) = {imm};
            d(index,2) = {[i j size]};
            index = index + 1;
        end
    end
end
```

Inizialmente (righe 2 - 4) viene definito il passo di sovrapposizione tra le immagini del dominio; di default la sovrapposizione è metà della dimensione dell'immagine. Successivamente (riga 10) si itera su ogni dimensione che si vuole assegnare al dominio e per ognuna di si va a definire lo step di sovrapposizione come `size / 1` (riga 11). I passi successivi (righe 12 e 13) sono quello di andare ad individuare tutti gli inizi di sottoimmagine del dominio (angolo in alto a sinistra dell'immagine), dato il valore di `step`, e quello prendere l'immagine di dimensione `size` (riga 15). Se questa sottoimmagine non è ancora presente nel dominio (riga 16), allora la si memorizza (riga 19), unitamente alle sue coordinate e alla sua dimensione (riga 20).

A questo punto abbiamo creato il dominio D che andremo ad utilizzare nei prossimi passi dell'algoritmo. Per comodità, siccome durante i confronti ogni immagine nel dominio viene dimezzata, al termine di questa procedura provvederemo a salvare una copia di ogni immagine del dominio di dimensioni dimezzate con opportuna trasformazione lineare.

3.2 Encoding

Ora si cerca di trovare una copertura ottimale per ogni partizione dell'immagine.

```
1  function [ partial_enc ] =
2      qtfunction(img, pos, size, split_t, min_size, max_size, doms, min_rms)
3
4      global img_copy;
5      x = pos(1);
6      y = pos(2);
7      partial_enc = [];
8
9      if size/2 > max_size
10         a = qtfunction(img, [x y], size/2, split_t,
11             min_size, max_size, doms, min_rms);
12         b = qtfunction(img, [x+size/2 y], size/2, split_t,
13             min_size, max_size, doms, min_rms);
14         c = qtfunction(img, [x y+size/2], size/2, split_t,
15             min_size, max_size, doms, min_rms);
16         d = qtfunction(img, [x+size/2 y+size/2], size/2, split_t,
17             min_size, max_size, doms, min_rms);
18         partial_enc = [a; b; c; d;];
19         return
20     end
21
22     rms = repmat(+Inf, [1 4]);
23     ts = zeros(1, 4);
24     doms_ind = zeros(1, 4);
```

```
25
26      for i = 1:length(doms)
27          d = doms{i,1};
28          if length(d) ~= size/2
29              continue
30          end
31
32          if rms(1) > min_rms
33              [r, t] = sup_dist(img(x:(x + size/2 -1),
34                                    y:(y + size/2 -1)), d);
35              if r < rms(1)
36                  rms(1) = r;
37                  ts(1) = t;
38                  doms_ind(1) = i;
39              end
40          end
41
42          if rms(2) > min_rms
43              [r, t] = sup_dist(img((x + size/2):(x+size-1),
44                                    y:(y + size/2 -1)), d);
45              if r < rms(2)
46                  rms(2) = r;
47                  ts(2) = t;
48                  doms_ind(2) = i;
49              end
50          end
51
52          if rms(3) > min_rms
53              [r, t] = sup_dist(img(x:(x + size/2 - 1),
54                                    (y + size/2):(y+size-1)), d);
55              if r < rms(3)
56                  rms(3) = r;
57                  ts(3) = t;
58                  doms_ind(3) = i;
59              end
60          end
61
62          if rms(4) > min_rms
63              [r, t] = sup_dist(img((x + size/2):(x+size-1) ,
64                                    (y + size/2):(y+size-1)) , d);
65              if r < rms(4)
66                  rms(4) = r;
67                  ts(4) = t;
68                  doms_ind(4) = i;
```

```
69              end
70          end
71      end
72
73      if rms(1) > split_t && size/2 > min_size
74          partial_enc = [partial_enc; qtfunction(img, [x y], size/2,
75               split_t, min_size, max_size, doms, min_rms)];
76      else
77          img1 = img(x:(x + size/2 -1), y:(y + size/2 -1));
78          [s, o] = least_squared_params(img1, doms{doms_ind(1),1});
79          partial_enc = [partial_enc;
80              [x y size/2 doms{doms_ind(1),2} ts(1) s o]];
81      end
82
83      if rms(2) > split_t && size/2 > min_size
84          partial_enc = [partial_enc; qtfunction(img, [x+size/2 y], size/2,
85              split_t, min_size, max_size, doms, min_rms)];
86      else
87          img1 = img((x + size/2):(x+size-1), y:(y + size/2 -1));
88          [s, o] = least_squared_params(img1, doms{doms_ind(2),1});
89          partial_enc = [partial_enc;
90              [x+size/2 y size/2 doms{doms_ind(2),2} ts(2) s o]];
91      end
92
93      if rms(3) > split_t && size/2 > min_size
94          partial_enc = [partial_enc; qtfunction(img, [x y+size/2], size/2,
95              split_t, min_size, max_size, doms, min_rms)];
96      else
97          img1 = img(x:(x + size/2 - 1), (y + size/2):(y+size-1));
98          [s, o] = least_squared_params(img1, doms{doms_ind(3),1});
99          partial_enc = [partial_enc;
100             [x y+size/2 size/2, doms{doms_ind(3),2} ts(3) s o]];
101     end
102
103     if rms(4) > split_t && size/2 > min_size
104         partial_enc = [partial_enc; qtfunction(img, [x+size/2 y+size/2], size/2,
105             split_t, min_size, max_size, doms, min_rms)];
106     else
107         img1 = img((x + size/2):(x+size-1) , (y + size/2):(y+size-1));
108         [s, o] = least_squared_params(img1, doms{doms_ind(4),1});
109         partial_enc = [partial_enc;
110             [x+size/2 y+size/2 size/2, doms{doms_ind(4),2} ts(4) s o]];
111     end
112 end
```

Come prima operazione si valuta se il partizionamento produce sottoimmagini sufficientemente piccole (riga 9) e, nel caso questo non avvenga, si procede a dividere in 4 parti l'immagine considerata e ad applicare ricorsivamente la procedura (righe 10 - 19).

Successivamente, per ognuna della quattro sottoimmagini generate tramite il partizionamento con *quadtree*, vengono calcolati i valore della distanza d e della rotazione dell'immagine del dominio. Ad esempio per la sottoimmagine in alto a sinistra la procedura parte col considerare tutti gli elementi del dominio, (riga 26) e per ognuno, tramite la funzione `sup_dist` (riga 33) verificare se viene trovato un valore di distanza d minore di quelli calcolati fino a quel momento. Per fare questa operazione viene considerata anche ogni possibile roto-traslazione dell'immagine stessa. Se il valore è effettivamente minore, questo viene memorizzato (riga 36) unitamente alle indicazioni sulla rotazione e all'indice di posizione dell'elemento del dominio (righe 37 e 38). Questa operazione viene ripetuta per tutte e quattro le sottoimmagini.

Ora che per ogni sottoimmagine è stata individuata la migliore copertura si passa a verificare se questa effettivamente soddisfava il valore di threshold voluto e di dimensioni minime. In caso positivo si memorizzano tutti i valori necessari, in caso contrario si applica ricorsivamente la funzione al quarto di immagine. Ad esempio per la sottoimmagine in alto a sinistra la procedura parte con il verificare se le soglie di threshold e dimensione sono rispettate (riga 73) e in caso negativo si riapplica la procedura a questo quarto di immagine (righe 74 - 75). Nel caso in cui i valori di soglia siano rispettati invece si calcolano i valori di `s` e di `o` (riga 78) e si memorizza il tutto nella mappa W (righe 79 e 80). Questa operazione viene ripetuta per tutte e quattro le sottoimmagini.

A questo punto l'encoding è terminato, in quanto si ha a disposizione la mappa W che, come abbiamo visto precedentemente, contiene tutte le informazioni che servono per trasformare un immagine qualsiasi nell'immagine di partenza.

3.3 Metrica

Nel precedente frammento di codice è stata introdotta la funzione `sup_dist` per il calcolo della distanza tra immagine nel dominio e immagine proveniente dal *quadtree*, di seguito è riportato il funzionamento effettivo di questa funzione, che rispecchia le equazioni viste in precedenza per il calcolo della distanza basandosi sui valori ottimali di s e o.

```
1  function [ rms, transf ] = sup_dist( img1, img2 )
2      rms_arr = zeros(1,8);
3
4      for i = 1:8
5          rms_arr(i) = dsh(img1, rotate_image(img2, i));
6      end
7
8      transf = find(rms_arr - min(rms_arr) == 0, 1);
```

```
9       rms = rms_arr(transf);
10  end
11
12  function [s, o] = least_squared_params(img1, img2)
13      n = length(img1)^2;
14      a = single(reshape(img2, [1, n]));
15      b = single(reshape(img1, [1, n]));
16
17      s = ( n * dot(a, b) - sum(a)*sum(b) ) / ( n * sum(a.^2) - sum(a)^2 );
18      s = min(s, 1);
19      o = ( sum(b) - s*sum(a) ) / n;
20  end
21
22  function [ rms ] = dsh(img1, img2)
23      [s, o] = least_squared_params(img1, img2);
24
25      n = length(img1)^2;
26      a = single(reshape(img2, [1, n]));
27      b = single(reshape(img1, [1, n]));
28
29      rms = sqrt(( sum(b.^2) + s * (s*sum(a.^2) - 2*dot(a,b)
30          + 2*o*sum(a)) + o*(n*o - 2*sum(b)) ) / n);
31  end
```

La funzione principale è `sup_dist`, che si occupa di effettuare il confronto tra la sottoimmagine presa in cosiderazione e tutte le possibili 8 rototraslazioni dell'elemento del dominio (righe 4 - 8). Questa funzione restituisce poi il valore della distanza e la rotazione, rappresentata da una mappa di valori da 1 a 8, (righe 10 e 11).

La seconda funzione, `least_squared_params`, si occupa, date due immagini, di trovare i valori di s e di o secondo le equazioni viste nel precedente capitolo (righe 17 - 19).

La terza funzione, `dsh`, si occupa, date due immagini e calcolati i valori di s e di o tramite la precedente funzione, di calcolare il valore della distanza (righe 29 e 30).

3.4 Output dell'esecuzione

Dopo aver introdotto gli algoritmi che abbiamo sviluppato vediamo brevemente quale è il risultato dell'encoding e del deconding. Prendiamo ad esempio un'immagine digitale, riportata in Figura 3.1, con il viso di una ragazza. L'immagine è in formato BITMAP a 8 bits in scala di grigi e ha dimensioni 512×512 pixel. Abbiamo scelto questa immagine perché ha delle vaste zone di colore uniforme, come il viso o lo sfondo, e altre zone estremamente complesse, come quelle coperte dai capelli. Questi dettagli ci hanno permesso di valutare la bontà della nostra implementazione.

Figura 3.1: girl grayscale, 128×128 pixels.

L'immagine è stata codificata con i seguenti parametri:

- `dom_range = [4, 8, 16, 32]`
- `l = 2`
- `min_rms = 4`
- `min_rms = 10`

L'output prodotto dal nostro algoritmo è una tabella di 36760×9 elementi, le cui dimensioni complessiva sono di circa 1MB. Il tempo di encoding è stato di circa 10 ore su una macchina con processore Pentium *i7 @ 2.4GHz* e 4 GB di RAM. Non applicando nessuna ottimizzazione o compressione della tabella stessa il peso rimane piuttosto alto.

Di seguito, in Figura 3.2, riportiamo la sequenza di immagini successive che si ottengono durante l'iterazione della procedura di decoding (ovvero l'applicazione della funzione W che definisce l'*iterated system* associato all'immagine di partenza). Ad ogni passo vengono applicate le trasformazioni descritte nella tabella prodotta, a partire da un'immagine completamente nera. Sono stati eseguiti complessivamente 12 step, ma già dal decimo in poi si può notare, anche a occhio, come le variazioni siano minime. Il risultato finale (quello raffigurato nell'ultima immagine) presenta dettaglio massimo che siamo riusciti ad ottenere con l'implementazione MATLAB qui sopra descritta. L'immagine non è stata ricostruita con un grande livello di dettaglio, ma dobbiamo osservare che la figura di partenza (quella scelta per testare l'implementazione) era estremamente complessa.

Figura 3.2: 12 step di decoding per l'immagine della ragazza.

4 Test, benchmarks

Come già sottolineato nell'introduzione, un aspetto importante da considerare durante la trattazione di tecniche di compressione "non ortodosse" è quello relativo alla loro effettiva usabilità in scenari realistici. È per questo motivo molto importante, una volta terminata la trattazione degli aspetti teorici della tecnica, effettuare una serie di test e benchmarks che consentano di effettuare una comparazione (che sia il più possibile oggettiva) tra i risultati ottenuti con la tecnica di compressione descritta e quelli ottenibili utilizzando invece una delle tecniche di compressione tra quelle più comunemente utilizzate.

Poiché l'implementazione MATLAB descritta nel precedente capitolo è stata sviluppata esclusivamente per fini "didattici", e pertanto senza che venisse posta particolare attenzione ad aspetti quali effettiva usabilità in contesti reali e prestazioni del codice su immagini di dimensioni realistiche, abbiamo scelto di effettuare i nostri test utilizzando una libreria di compressione basata su frattali che fosse, oltre che più flessibile, anche più performante del codice da noi sviluppato. La nostra scelta è ricaduta su *Fiasco* [1] [2] [3], una libreria *open source* di compressione frattale implementata in C.

Dato che la compressione tramite frattali è inerentemente *lossy*, la scelta naturale per il formato con cui effettuare la comparazione è rappresentata dallo standard JPEG[4]. Sono stati effettuati test di due differenti tipi: una prima batteria di test pensata per valutare la capacità delle due tecniche di compressione di gestire immagini di varie dimensioni (anche molto grandi), anche valutando i tempi di codifica e decodifica, ed una seconda batteria di test in cui si è cercato di effettuare una comparazione delle qualità delle immagini decompresse.

4.1 Efficienza ed efficacia della codifica

Per valutare efficienza (ovvero la rapidità della codifica) ed efficacia (ovvero i rapporti di compressione ottenuti) della compressione frattale e della compressione JPEG, sono state utilizzate sette versioni, differenti in dimensioni, di una immagine campione raffigurante un paesaggio naturale. Nella tabella seguente sono riportati i nomi, le dimensioni ed il peso delle immagini utilizzate. Tutte le immagini sono fotografie in *grayscale*, salvate come file in formato `.pgm`.

In un primo test abbiamo cercato di valutare la effettiva capacità delle due librerie di comprimere le immagini di test, e i rapporti di compressione ottenuti, lasciando in un primo momento da parte la valutazione dell'efficienza dell'operazione di codifica (ovvero

Immagine	Dimensioni	Peso
`california_coast_320`	320 × 240	0.07 MB
`california_coast_640`	640 × 480	0.30 MB
`california_coast_800`	800 × 600	0.48 MB
`california_coast_1200`	1200 × 900	1.10 MB
`california_coast_1600`	1600 × 1200	1.90 MB
`california_coast_2400`	2400 × 1800	4.30 MB
`california_coast_3648`	3648 × 2736	10.0 MB

Figura 4.1: Le immagini di test

il tempo impiegato per comprimere l'immagine). I risultati del test sono riportati in tabella

Immagine	JPEG		Fiasco	
	Peso	Compressione	Peso	Compressione
`california_coast_320`	026.5 kB	2.64x	02.7 kB	25.9x
`california_coast_640`	100.9 kB	2.97x	10.5 kB	28.5x
`california_coast_800`	147.3 kB	3.25x	16.3 kB	29.4x
`california_coast_1200`	330.4 kB	3.32x	✗	
`california_coast_1600`	515.5 kB	3.68x	✗	
`california_coast_2400`	1.00 MB	4.30x	✗	
`california_coast_3648`	1.90 MB	5.26x	✗	

Figura 4.2: Pesi e rapporti di compressione delle immagini codificate

Mentre la libreria JPEG è in grado di comprimere senza nessuna difficoltà le immagini utilizzate per il test, Fiasco fallisce sulle ultime 4 immagini con un `Maximum number of states reached!` error. Considerando che il primo file che Fiasco non riesce a trattare è una modesta immagine 1200 × 900 del peso di 1.1 MB, questo test rende abbastanza evidente come la compressione tramite frattali sia di fatto eccessivamente costosa (da un punto di vista computazionale) per permetterne l'utilizzo con immagini ad alta risoluzione.

D'altra parte, sulle immagini che entrambe le librerie hanno potuto comprimere, appare evidente che la compressione tramite frattali consente di ottenere rapporti di compressione molto competitivi (un ordine di grandezza meglio del JPEG in questo caso). Va tuttavia sottolineato che il fattore di compressione andrà valutato anche sulla base dei tempi di codifica e della qualità dell'immagine ricavata successivamente, in fase di decompressione.

I tempi di compressione sono riportati nella tabella seguente:

Appare ora evidente il motivo per cui Fiasco ha rinunciato a comprimere le immagini più grandi: l'operazione di codifica è estremamente costosa, e i tempi di codifica vanno

Immagine	JPEG	Fiasco	Rapporto
`california_coast_320`	0.006s	0.549s	91.5x
`california_coast_640`	0.010s	8.620s	861x
`california_coast_800`	0.016s	18.51s	1156x
`california_coast_1200`	0.028s	✗	–
`california_coast_1600`	0.042s	✗	–
`california_coast_2400`	0.087s	✗	–
`california_coast_3648`	0.183s	✗	–

Figura 4.3: I tempi di codifica

peggiorando esponenzialmente. JPEG comprime invece tutte le immagini testate in tempi molto rapidi.

4.2 Qualità delle immagini

Non è facile, in generale, valutare in maniera oggettiva la qualità di due immagini compresse con una tecnica *lossy* e successivamente decompresse. In questo caso, tuttavia, la differenza tra le immagini ricavate da JPEG e Fiasco è significativa, e visibile ad occhio nudo.

Figura 4.4: Da JPEG e da Fiasco, rispettivamente, dopo la decompressione

Andando a zoomare su un particolare dell'immagine, appare evidente come il rapporto di compressione ottenuto da Fiasco sia ottenuto al costo di una grossa perdita di dettaglio. Se avessimo deciso di codificare le immagini JPEG con una minore qualità (controllabile in fase di encoding tramite un parametro standard), probabilmente anche queste ultime avrebbero presentato un minor livello di dettaglio, ma il rapporto di compressione sarebbe aumentato notevolmente. Per quanto concerne questo test, abbiamo deciso di codificare le immagini in alta qualità, lasciando il parametro di input al valore standard applicato dalle librerie comunemente utilizzate.

Figura 4.5: Da JPEG e da Fiasco, rispettivamente, dopo la decompressione (dettaglio)

4.3 Conclusioni

In questo capitolo abbiamo confrontato efficienza, efficacia e risultato finale della codifica di immagini tramite frattali e della codifica tramite lo standard JPEG. Sulle immagini che sono effettivamente alla portata di un metodo di compressione frattale, Fiasco consente di effettuare una compressione di circa un ordine di grandezza migliore di quella garantita da JPEG. Nonostante la qualità finale delle immagini dopo la decodifica sia non eccellente (rispetto a quella JPEG), l'aspetto delle immagini ricostruite è buono, soprattutto considerando il rapporto di compressione raggiunto. Un'osservazione attenta dei particolari delle immagini rivela una buona perdita di qualità, ma le immagini, nel loro complesso, non presentano artefatti troppo evidenti.

Il costo computazionale del processo di codifica è senza dubbio il punto debole delle tecniche di compressione frattale. Mentre il tempo necessario per la compressione JPEG scala in maniera lineare con le dimensioni dell'immagine, quello impiegato da Fiasco aumenta in maniera esponenziale all'aumentare delle dimensioni, arrivando ad essere di 3 ordini di grandezza peggiore per un'immagine dalle (modeste) dimensioni di 1200 $\times$ 900. Tale inefficienza previene inoltre l'utilizzo della tecnica con immagini in alta definizione, che sono semplicemente fuori dalla portata delle tecniche di compressione

frattale (anche rilassando i requisiti di qualità).

Appendice A

Tutorial per l'esecuzione del codice sorgente

In questa appendice diamo una breve descrizione di come utilizzare il codice sorgente con cui sono stati realizzati gli esempi e i bechmarks utilizzati nel corso dell'esposizione.

Per l'esecuzione del codice MATLAB è necessario aver installata l'*Image Processing Toolbox*[1].

Per l'esecuzione dei benchmarks è necessario che siano installati sul sistema *ImageMagick*[2], libreria per la manipolazione di immagini utilizzata per effettuare la compressione in formato JPEG, e Fiasco[3], libreria sperimentale per la compressione di immagini utilizzando FIC.

La prima cosa da fare è clonare il repository che contiene tutti i sorgenti:

```
git clone https://github.com/luca-dex/shiny-octo-dubstep
```

Il contenuto della cartella è organizzato nel seguente modo:

- `src/` contiene i sorgenti dell'algoritmo implementato in MATLAB
- `benchmarks` contiene gli script per eseguire i benchmarks
- `doc/` contiene il sorgente da cui è stato realizzato questo documento
- `workspace/` Ccontiene esempi di immagini compresse realizzate con l'algoritmo implementato in MATLAB

A.1 Implementazione MATLAB

Il codice MATLAB è progettato per essere eseguito in maniera parallela su una macchina dotata di quattro cores. La prima parte del codice effettua l'encoding dell'immagine. L'impostazione di tutti i parametri iniziali avviene all'interno del file `main.m`. I parametri che è possibile configurare sono:

[1] `mathworks.com/products/image/`
[2] `www.imagemagick.org/`
[3] `github.com/l-tamas/Fiasco`

- `im`: Path dell'immagine da comprimere, che deve essere quadrata ed in formato BMP
- `dom_range`: Dimensione degli elementi che verranno salvati nel dominio. Devono essere divisori della lunghezza del lato dell'immagine
- `l`: Parametro di sovrapposizione. Il default è 2
- `min_rms`: La soglia di errore oltre la quale si accetta la copertura selezionata
- `max_range`: La soglia di errore oltre la quale si rifiuta la copertura selezionata e si riapplica la suddivisione in 4 parti
- `output`: Path dell'output dell'algoritmo (workspace MATLAB).

Una volta che tutti i parametri sono stati configurati, l'esecuzione può essere lanciata. Il tempo di calcolo è molto alto, poiché il codice non è stato sviluppato in maniera ottimizzata. È consigliato non eccedere con la dimensione delle immagini (512×512 è il limite oltre il quale il codice non è stato testato).

Una volta che l'immagine è stata compressa, può essere ricostruita con la funzione `decode_all`. Dentro a questa funzione deve essere configurato il parametro `workspace`, con il path del workspace salvato dalla funzione precedente. Questa funzione mostra per prima cosa l'immagine originale e, ogni volta che viene effettuato un click del mouse all'interno dell'immagine, mostra lo step di ricostruzione successivo.

A.2 Benchmarks

La *repository* contiene anche gli script bash che sono stati utilizzati per i test (cartella `benchmarks`). Le immagini di test sono contenute nella sotto-cartella denominata `images`. Per lanciare i test, è necessario eseguire i seguenti script, in ordine:

1. `convert2pgm.sh`, che converte le immagini di test in formato `pgm`, l'unico supportato da Fiasco
2. `encode2fiasco.sh`, che comprime le immagini `pgm` create al passo precedente utilizzando Fiasco
3. `encode2jpeg.sh`, che comprile le immagini di test utilizzando JPEG
4. `decodefiasco.sh`, che decodifica e decomprime le immagini compresse tramite frattali

Tutti i risultati delle computazioni vengono salvati in una propria cartella, che i vari script si occupano di creare e popolare. La velocità delle operazioni di compressione è misurata in automatico dai relativi script utilizzando il comando UNIX `time`.

Gli script sono stati sviluppati, testati ed eseguiti in ambiente Linux.

Bibliografia

[1] Ullrich Hafner *Fractal Image And Sequence COdec.* `https://github.com/l-tamas/Fiasco`.

[2] Ullrich Hafner, Juergen Albert, Stefan Frank, and Michael Unger *Weighted Finite Automata for Video Compression* IEEE Journal on Selected Areas In Communications, January 1998.

[3] Ullrich Hafner *Low Bit-Rate Image and Video Coding with Weighted Finite Automata* Ph.D. thesis, Mensch & Buch Verlag, ISBN 3-89820-002-7, October 1999.

[4] IEEE *The JPEG Standard - ISO/IEC 10918*