Chi ha detto che non si sbircia tra i frattali con gli occhi del PostScript?

Marcello Seri, Alice Venturini Barazzuol

13 dicembre 2006

Indice

1 La geometria frattale **3**
1.1 Introduzione 3
1.2 Il triangolo di Sierpinski 5
1.3 Il Merletto di Koch 6
1.4 Frattali L-System 6
1.5 Successioni di insiemi piani 8

2 Un'introduzione a PostScript **10**
2.1 Il Linguaggio PostScript 10
2.2 Disegnare cammini 11
2.3 Operazioni Aritmetiche 12
2.4 Variabili e funzioni 13
2.5 Gli array 16
2.6 I cicli 17
2.7 Istruzioni condizionali 18

3 Disegnamo qualche frattale in PostScript **19**
3.1 Il codice del Drago 19
3.1.1 Semplici esperimenti sul codice di *dragon.ps* 22
3.1.2 Semplici esperimenti sul codice di *ls-filo_ erba.ps* 24
3.2 Le trasformazioni della felce ed altri suoi simili... 25
3.2.1 Semplici esperimenti sul codice di *felce.ps* 26

Bibliografia **28**

Preambolo

La matematica è affascinante. Apre di continuo finestre su mondi immensi, inesplorati, incomprensibili e spesso incredibili!

Per molti è solo un'arida accozzaglia di astrattismi inutili buttati là alla rinfusa... ma sanno bene che molte delle cose che li circondano e che usano quotidianamente funzionano grazie alla matematica!

Una delle tante critiche che mi sento rivolgere spesso, riguarda l'assurdità di voler rappresentare il mondo con solidi regolari, curve geometriche "lisce" e linee dritte... sono tutte astrazioni lontane dalla realtà! Per fortuna si può controbattere e si può farlo in fretta e in modo, molto spesso, stupefacente!!!

I frattali (dal latino *fractus*: frastagliato) sono delle curve geometriche che riescono a rappresentare la straordinaria varietà, irregolarità e caoticità della natura in maniera sorprendentemente facile e veritiera!

Con questo lavoro vogliamo, seppur di sfuggita, sollevare il velo che nasconde questo mondo di matematica sconosciuto ai più. Quale modo migliore per cercare di avvicinarsi al PostScript? Un linguaggio pensato per la stampa di immagini e testi impaginati...

In questo piccolo ma intenso testo abbiamo provato a rappresentare alcuni semplici frattali sfruttando, appunto, il linguaggio PostScript ed alcuni algoritmi semplici ma dagli ottimi risultati.

Per il momento accontentiamoci di sapere che in questo modo potremmo lasciare il gravoso compito di calcolare la curva e rappresentarla alla stampante!!!

Capitolo 1

La geometria frattale

1.1 Introduzione

Durante una passeggiata in campagna o in un bosco siamo immersi nella natura, fra montagne, alberi, erbe e fiori di tutti i tipi e di tutte le dimensioni. A parte l'indiscutibile bellezza dell'ambiente, un occhio più esperto può cogliere nella forma di tutti questi oggetti delle curiose proprietà geometriche.

Le forme che si incontrano però non possono essere studiate applicando gli assiomi della semplice geometria euclidea. Infatti, non si tratta (tranne pochissime eccezioni) di enti geometrici nel senso euclideo del termine, ovvero di poligoni o poliedri più o meno regolari. Tutto cio' che si incontra in natura è molto più complesso, frammentato, frastagliato per quanto sia intrinsecamente impregnato di regolarità e simmetrie. Questo ha giustificato l'introduzione di un nuovo tipo di geometria da parte del matematico Benoit B. Mandelbrot (1982): la geometria frattale.

Figura 1.1: Una foglia di Felce e le sue autosimilarità

Consideriamo ad esempio una comune felce (Fig. 1.1). La cosa che si nota immediatamente è che una parte della felce è simile a tutta la felce stessa, ovvero è una copia in piccolo della foglia completa.

Ora concentriamoci, invece, su un insieme di N trasformazioni (non necessariamente affini) del piano cartesiano: $\{T_1, T_2, T_3, \ldots, T_N\}$ ed applichiamole allo stesso sottoinsieme A del piano. Come risultato otterremo una famiglia di N sottoinsiemi del piano cartesiano $\{T_1(A), T_2(A), T_3(A), \ldots, T_N(A)\}$.

Sia A_1 l'insieme ottenuto come unione di questi sottoinsiemi. Applichiamo di nuovo le N trasformazioni all'insieme A_1 così ottenuto e consideriamo l'unione degli N insiemi immagine. Chiamiamo questo insieme A_2 . Agiamo nello stesso modo su A_2 e otteniamo A_3. Continuando allo stesso modo, otteniamo una successione di insiemi $\{A_1, A_2, A_3, \ldots\}$.

Il problema che ci poniamo è il seguente: la successione $\{A_n\}$ di insiemi converge? Convergere, in questo caso, vuol dire che la successione si stabilizzerà, e, da un certo punto in poi, non noteremo più cambiamenti apprezzabili nell'immagine sullo schermo.

Sotto certe condizioni la successione di insiemi convergerà ad un insieme limite F. Questo insieme limite F si definisce frattale, anzi **frattale IFS (Iterated Function System)** ovvero frattale ottenuto iterando un insieme di trasformazioni del piano.

Tuttavia esistono altri tipi di frattali, fra cui i **Frattali LS** che rivestono una particolare importanza, soprattutto in biologia.

LS sta per **Lindenmayer-System** dal nome dello studioso che li ha introdotti negli anni '60 per descrivere i fenomeni di crescita delle piante. *I Frattali LS possono essere visti come una generalizzazione di quelli IFS.*
Vediamo un esempio di frattale LS che però non è un frattale IFS: il **Fiocco**

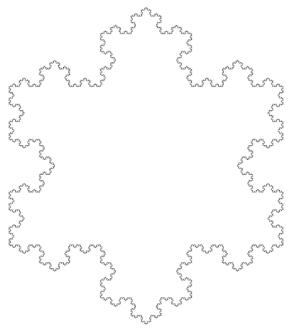

Figura 1.2: Il fiocco di neve di Koch

di neve, o merletto di Koch. Non è un frattale IFS poiché non è possibile suddividere la figura in un numero di "sottocopie" esatte della figura stessa: ovvero non è completamente autosimile. D'altra parte è possibile, però dividere la figura in tre copie del merletto di Koch, che è un frattale IFS.

1.2 Il triangolo di Sierpinski

Cerchiamo il minimo numero di copie del frattale che permetta di rappresentarlo tutto. Devono essere ricoperti tutti i punti del frattale e ciascun punto deve essere scelto una sola volta. In altre parole, è come se dovessimo ricoprire il frattale con tante copie del frattale stesso, in modo tale che ciascun punto sia coperto una sola volta. Le copie sono 3 e sono evidenziate in figura (Fig. 1.3).

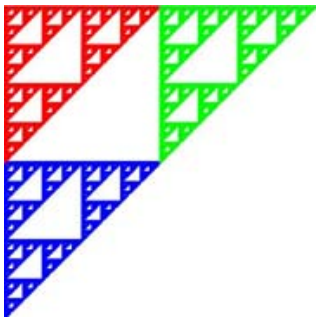

Figura 1.3: Il triangolo di Sierpinski

Ora cerchiamo le trasformazioni geometriche che applicate al frattale, lo trasformano nelle tre copie che abbiamo individuato. Abbiamo tre copie e di conseguenza cerchiamo tre trasformazioni. Si tratta di tre omotetie di ragione 1/2 poiché la figura diventa la metà di quella iniziale. Poi occorrerà comporle con opportune traslazioni.

Per ottenere un'espressione analitica delle trasformazioni, occorre fissare un opportuno sistema di riferimento. Per semplicità supponiamo che il frattale sia costruito dentro il quadrato di lato unitario e l'origine sia posta nell'angolo a sinistra in basso.

Per ottenere il triangolo di Sierpinski basta usare le seguenti tre trasformazioni:

$$T_1 : \begin{cases} X = \frac{1}{2}x; \\ Y = \frac{1}{2}y \end{cases} \quad T_2 : \begin{cases} X = \frac{1}{2}x; \\ Y = \frac{1}{2}y + \frac{1}{2} \end{cases} \quad T_3 : \begin{cases} X = \frac{1}{2}x + \frac{1}{2}; \\ Y = \frac{1}{2}y + \frac{1}{2} \end{cases}$$

L'origine del sistema di riferimento è posta nel vertice in basso a sinistra del quadrato di partenza. Notare che T_1 è un'omotetia di ragione $\frac{1}{2}$, T_2 è

un'omotetia di ragione $\frac{1}{2}$ composta con una traslazione secondo il vettore $(0, \frac{1}{2})$, T_3 è un'omotetia di ragione $\frac{1}{2}$ composta con una traslazione secondo il vettore $(1/2, 1/2)$.

1.3 Il Merletto di Koch

Anche in questo caso determiniamo il numero di copie che ci permette di ricoprire tutto il frattale senza sovrapposizioni. È facile verificare che il numero delle le copie in questo caso è 4 e tale sarà anche il numero delle trasformazioni geometriche che generano il frattale (vedi Fig. 1.2).

Le trasformazioni sono le seguenti:

$$T_1 : \begin{cases} X = \frac{1}{3}x; \\ Y = \frac{1}{3}y \end{cases} \quad T_2 : \begin{cases} X = \frac{1}{6}x - \frac{\sqrt{3}}{6}y + \frac{1}{3}; \\ Y = \frac{\sqrt{3}}{6}x + \frac{1}{6}y \end{cases}$$

$$T_3 : \begin{cases} X = -\frac{1}{6}x + \frac{\sqrt{3}}{6}y + \frac{1}{3} + \frac{2}{3}; \\ Y = \frac{\sqrt{3}}{6}x + \frac{1}{6}y \end{cases} \quad T_4 : \begin{cases} X = \frac{1}{3}x + \frac{2}{3}; \\ Y = \frac{1}{3}y \end{cases}$$

1.4 Frattali L-System

E' molto interessante, nell'ambito dell'esplorazione delle curve frattali, l'utilizzo del semplice linguaggio introdotto da Lindenmayer (noto come L-System) che fa uso di pochi simboli che possono essere riassunti nei seguenti, a fianco dei quali è riportata l'azione a loro collegata:

- `F` : traccia un segmento nella direzione attuale
- `+` : gira a destra (di un angolo prefissato)
- `-` : gira a sinistra (dello stesso angolo)
- `[` : memorizza l'attuale posizione e direzione
- `]` : torna al punto (e alla direzione) memorizzato

Tale linguaggio, come vedremo nei nostri esperimenti può essere esteso o semplificato ulteriormente. Sta di fatto che definendo la figura di partenza, l'angolo di rotazione e la legge di ricorrenza, si possono ottenere figure estremamente interessanti.

Un semplice esempio è dato dal seguente codice (cfr. *ls-filo_ erba.ps*)

```
0=F
a=30
F=F[+F]F[-F]F
```

viene così interpretato:

- si parte dalla stringa `F`
- l'angolo di rotazione è 30 gradi ($\pi/6$)
- si utilizza una legge ricorsiva che ad ogni iterazione sostituisce ad ogni `F` nella stringa, la nuova stringa `F=F[+F]F[-F]F`

Pertanto, dopo la prima iterazione, la stringa iniziale `F` diventa

`F[+F]F[-F]F`

ovvero

- `F` : tracciamo un segmento (verticale);
- `[` : memorizziamo questo punto e la direzione verticale;
- `+F`: giriamo a destra e tracciamo un segmento;
- `]` : torniamo al punto memorizzato;
- `F` : tracciamo un segmento verticale;
- `]` : memorizziamo questo punto e la direzione verticale;
- `-F`: giriamo a sinistra e tracciamo un segmento;
- `]` : torniamo al punto memorizzato;
- `F` : tracciamo un segmento verticale.

Se ora facciamo una seconda iterazione, risostituendo ad ogni `F` in `F[+F]F[-F]F` la stringa `F[+F]F[-F]F`, (cioè sostituendo ad ogni segmento una figura simile a quella ottenuta) otteniamo
`F[+F]F[-F]F [+F[+F]F[-F]F] F[+F]F[-F]F [-F[+F]F[-F]F] F[+F]F[-F]F`
(gli spazi servono solo a rendere più evidenti le sostituzioni).

Eseguendo i comandi contenuti in tale stringa si ottiene la seconda figura, ed iterando il procedimento la stringa diventa sempre più lunga e la figura da essa generata sempre più simile ad un oggetto reale, in questo caso un filo d'erba (Fig.1.4).

Modificando tale codice si possono ottenere esempi più realistici ed accattivanti, come vedremo più avanti.

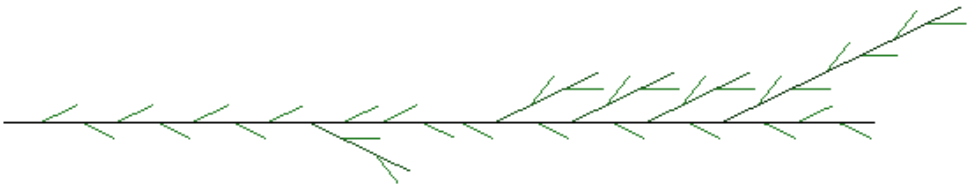

Figura 1.4: Il risultato ottenuto da questo primo esempio

1.5 Successioni di insiemi piani

Possiamo fare un altro passo avanti, occupandoci delle successioni di insiemi nel piano.

Consideriamo una semplice funzione dal piano cartesiano (x, y) al piano cartesiano (X, Y), definita da:

$$f : \begin{cases} X = ax + by + u \\ Y = cx + dy + v \end{cases} \tag{1.1}$$

Tale funzione è lineare (ci limiteremo a tale caso per semplicità), cioè le variabili compaiono in modo lineare (x ed y sono moltiplicate per una costante e sommate con un altra costante; non vi sono potenze o funzioni complicate).

Come si può notare facendo due calcoli il punto ($x = 0$, $y = 0$) viene trasformato nel punto ($X = u$, $Y = v$); così il punto ($x = 1$, $y = 0$) va nel punto ($X = a+u$, $Y = c+v$); il punto $(0, 1)$ va in $(b+u, d+v)$, il punto $(1, 1)$ va in $(a + b + u, c + d + v)$; cioè i punti interni in quadrato nel piano (x, y) finiscono nei punti interni ad un parallelogramma nel piano (X, Y), ecc.

Possiamo quindi dire che la funzione che trasforma il punto (x, y) nel punto (X, Y) è una trasformazione di insiemi del piano in insiemi del piano (ad esempio il quadrato è stato trasformato nel parallelogramma).

Ora, se la distanza tra i trasformati $f(A)$ ed $f(B)$ di due punti qualunque A e B è minore della distanza che vi era tra gli stessi A e B, possiamo dire che la funzione in oggetto è una contrazione. In altre parole, la figura trasformata risulta più piccola della figura originale. Ci possiamo quindi aspettare che, se prendiamo un insieme qualunque ed applichiamo su di esso tante volte la trasformazione precedente, gli insiemi ottenuti diventino sempre più piccoli e tendano a diventare un punto (il punto fisso per la trasformazione in questione).

Non molto diverso è il caso in cui si usano insieme più di una contrazione.

Il risultato sarà l'unione dei punti fissi delle delle contrazioni e porterà a delle immagini a dir poco strabilianti. Vedremo alcuni esempi in seguito.

Capitolo 2

Un'introduzione a PostScript

2.1 Il Linguaggio PostScript

Il PostScript è un linguaggio di programmazione creato dall'Adobe nel 1985 ed ottimizzato per la stampa di immagini grafiche e testo. Propriamente è un linguaggio di descrizione di documento e come tale viene direttamente interpretato dalle stampanti laser.

Per semplificarne l'implementazione a livello hardware, il codice sfrutta la Notazione Polacca Inversa che, come vedremo, fa del suo legame con il concetto di stack (pila), il suo punto di forza.

Tramite il file PostScript, si viene a creare l'elenco di tutti gli oggetti grafici, che vengono descritti vettorialmente, rasterizzati dalla stampante e poi passati su carta attraverso gli inchiostri. E tutto viene fatto dalla stampante, alleggerendo il carico del computer che può essere usato per altri lavori.

D'altro canto si può sfruttare la stampante come calcolatore... come vedremo, i frattali che andremo a rappresentare, sono generati da un vero e proprio programma ricorsivo che viene eseguito nella stampante.

Si lavora con semplici file ASCII che contengono impostazioni per la stampante (il cassetto o il formato di carta da usare), definizioni relative al tipo di impaginazione, al tipo di font e alle sue dimensioni, direttive di comandi grafici vettoriali e immagini bitmap.

Arrivato, ormai, alla sua terza edizione (o livello, secondo la definizione Adobe), il linguaggio PostScript è oggi in grado di descrivere anche colori in quadricromia CMYK (nella sua prima versione descriveva solo scale di grigio e colori RGB) ed ha raggiunto una maturità tale da ottenere stampe di ottima qualità anche a livello professionale.

Il PostScript è sicuramente un linguaggio complesso e innaturale, quindi non riscuote molto successo come linguaggio di programmazione in sé (sebbe-

ne, ad esempio, NEXT era un sistema operativo che sfruttava pesantemente il PostScript per gestire sia la grafica a schermo che le stampanti...).

2.2 Disegnare cammini

Il disegno di oggetti in PostScript avviene attraverso la costruzione di *cammini*, ovvero una sequenza di linee e curve. Per iniziare un nuovo cammino si usa il comando `newpath`.

Per iniziare a disegnare il cammino, bisogna dapprima posizionarsi sul punto di partenza attraverso il comando `moveto`; mentre per creare una linea del cammino si usa il comando `lineto`, preceduto sempre dalle coordinate del punto di arrivo. Il cammino viene disegnato, ovvero è reso visibile, con il comando **stroke**.

Per esempio, la sequenza di comandi

```
newpath
144 144 moveto
288 144 lineto
288 288 lineto
144 288 lineto
144 144 lineto
stroke
```

disegna un quadrato di coordinate $(144, 144), (288, 144),$
$(144, 288), (288, 288)$ sul foglio.
Ci sono anche altri modi per disegnare questo quadrato; nell'esempio

```
newpath
144 144 moveto
288 0 rlineto
0 288 rlineto
-144 0 rlineto
closepath
stroke
```

i comandi `rlineto` e `rmoveto` indicano movimenti relativi al punto precedente, mentre `closepath` chiude il cammino nell'ultimo punto a cui è applicato il comando `moveto`.

Se invece si vuole colorare il cammino, si può usare il comando `x x x setrgbcolor`, dove `rgb` sta per *red, green, blue*, e per ogni colore si sceglie un gruppo di tre parametri compresi tra 0 e 1 (0 indica nero, 1 bianco). Per esempio

```
1 0 0 setrgbcolor
newpath
144 144 moveto
144 0 rlineto
0 44 rlineto
-144 0 rlineto
closepath
fill
```

disegna il quadrato mostrato nel precedente esempio tutto riempito di rosso.

Osservazione 2.1. *Dagli esempi si nota subito che i comandi* `moveto` *e* `lineto` *vengono sempre preceduti dalle coordinate dei punti, rispettivamente di partenza e di arrivo, del cammino. Questa convenzione viene chiamata* Notazione Polacca Inversa *e viene usata sempre in PostScript.*

2.3 Operazioni Aritmetiche

PostScript è un linguaggio di programmazione completo. Tuttavia le sue capacità aritmetiche sono abbastanza limitate. I numeri interi devono appartenere all'intervallo $[-32784, 32783]$, mentre i numeri reali hanno solo 7 cifre significative.

Per compiere tutte le operazioni, PostScript usa degli **stack**, vettori di lunghezza arbitraria. Il primo termine in input viene immesso alla fine dello stack, mentre l'ultimo viene posto all'inizio. Così, per sommare i numeri 3 e 4 si dovrà scrivere: 3 4 add.

$$\begin{cases} \textit{input} & \textit{stack} \\ 3 & 3 \\ 4 & 3\ 4 \\ \texttt{add} & 3\ 4\ \text{add} \\ = & 7 \end{cases}$$

Scrivendo il simbolo di uguaglianza vengono cancellati i dati precedentemente immessi nello stack e rimane solo il risultato dell'operazione, nell'esempio 7.

Ecco un elenco di comandi per le operazioni:

$x\ y$ `add`	inserisce nello stack $x + y$
$x\ y$ `sub`	inserisce nello stack $x - y$
$x\ y$ `mul`	inserisce nello stack xy
$x\ y$ `div`	inserisce nello stack $\frac{x}{y}$
$x\ y$ `idiv`	inserisce nello stack il quoziente di $\frac{x}{y}$
$x\ y$ `mod`	inserisce nello stack il resto di $\frac{x}{y}$
x `neg`	inserisce nello stack $-x$
$x\ y$`atan`	inserisce l'angolo in gradi compreso tra $x,\ y$
x `sqrt`	inserisce nello stack $\sqrt{x}$
x `sin`	inserisce nello stack $\sin x$
x `cos`	inserisce nello stack $\cos x$
$y\ x$ `exp`	inserisce nello stack y^x
x `ln`	inserisce nello stack $\ln x$
x `abs`	inserisce nello stack $\|x\|$
x `round`	inserisce nello stack l'intero più vicino ad x

2.4 Variabili e funzioni

Si è visto negli esempi precedenti come disegnare un quadrato di vertici $(144, 144), (144, 288), (288, 288), (288, 1444)$. Analizziamo ora il problema di disegnare un quadrato il cui lato è lungo 1.

A tale proposito si introduce la nozione di variabile.

Una variabile in PostScript viene definita in questo modo:

/nomevariabile valore `def`

nell'esempio in questione:

/`s` 1 `def`

Usando la variabile `s` per il lato del quadrato, il nuovo programma sarà:

```
/s 1 def
newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke
```

Può anche capitare di dover ripetere determinati comandi più di una volta nello stesso codice. È per questo motivo che vengono utilizzate le *procedure*.

Una procedura in PostScript è un insieme di comandi racchiuso da parentesi graffe { }. Ogni volta che nel codice verrà ripetuto il nome della procedura, si eseguiranno i comandi descritti nella procedura stessa.

Per esempio:

```
%!
/s 1 def
newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke
0 -1 translate
newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke
showpage
```

disegna due quadrati di lato 1, l'uno sotto l'altro. Introducendo la procedura

```
/draw-square
{ newpath
0 0 moveto
s 0 rlineto
0 s rlineto
s neg 0 rlineto
closepath
stroke
} def
```

basterà richiamare la procedura all'interno del codice, e si otterrà lo stesso effetto:

```
draw-square
0 -1 translate
draw-square.
```

Si supponga ora di voler disegnare due quadrati di lati di lunghezza differente. Il codice potrebbe essere di questo tipo:

```
/s 1 def
draw-square
/s 3 def
draw-square
```

Per ottenere un risultato analogo, ma in modo alternativo, si può definire la variabile `s` all'interno della definizione della procedura (mi si scusi il bisticcio). Questo problema si risolve facilmente inserendo nelle istruzioni della procedura lo statement:

```
/s exch def
```

All'interno del programa quindi si potrà richiamare la procedura in questo modo:

```
2 draw-square
```

Quello che succede è di facile interpretzione: sostanzialmente ponendo l'istruzione `exch` la procedura ha accesso al valore della variabile `s` posto nello stack, prima che la procedra stessa venga chiamata; di fatto il comando `exch` scambia i primi due termini posti nello stack.

Così, per disegnare un rettangolo con base e altezza differenti, si può creare una procedura di questi tipo:

```
/draw-rectangle {
/h exch def
/b exch def
newpath
0 0 moveto
b 0 rlineto
0 h rlineto
b neg 0 rlineto
closepath
stroke
} def
```

per poi richiamarla all'interno del programma, per esempio così:

2 3 `draw`-rectangle.

Una procedura può anche essere una funzione. Ovvero può prendere dei dati in input, per poi restituire nello stack il risultato dei calcoli elencati nelle istruzioni di definizione della procedura stessa.

Per esempio se volessimo calcolare la lunghezza della diagonale del rettangolo disegnato dalla procedura `draw-rectangle`, si potrebbe procedere in questo modo: scrivere la procedura `diagonale`:

```
/diagonale {
/a exch def
/b exch def
 a dup mul b dup mul add sqrt
} def
```

Tuttavia in PostScript è conveniente riscrivere la proceduracosì:

```
/diagonale {
     dup mul  a,  b × b
        exch  b × b,  a
     dup mul  b × b,  a × a
         add  b × b + a × a
        sqrt  √(b × b + a × a)
        }def
```

2.5 Gli array

I vettori in PostScript sono collezioni unidimensionali di oggetti che vengono indicizzate a partire da 0, in modo che, in un array dieci elementi, la numerazione vada da 0 a 9.

Un array in PostScript è definito da una qualsiasi collezione di oggetti delimitato da parentesi quadrate. Negli esempi:

```
[16 (ciao) 8]
```

è un vettore con tre elementi: due numeri ed un stringa; mentre

```
[(somma) 16 4 add]
```

è un vettore con due elementi: la stringa somma e il numero 20.

Per inserire dati in un array esiste il comando `put`. Per esempio, il frammento di codice

```
/AnArray 10 array def
AnArray 8 (language) put
```

pone la stringa language nella nona posizione del vettore `AnArray`.

Invece l'operatore `get` prende da input un array e un numero intero (l'indice) dallo stack e restituisce il valore della variabile che occupa la posizione indicata dall'indice nell'array. Nell'esempio:

```
[2 5 9] 1 get.
```

2.6 I cicli

Sarà certamente utile imparare ad usare i cicli, per riuscire ad iterare alcune operazioni all'interno del codice.

Iniziamo quindi con il loop più semplice, il ciclo `repeat`. L'implementazione avviene secondo questo schema: preso `N` intero,

```
N {
istruzione 1
istruzione 2
... ... ...
istruzione k
} repeat
```

Quello che succede è che le k istruzioni definite all'interno delle parentesi vengono ripetute `N` volte.

Passiamo ora alla descrizione del ciclo `for`. Il ciclo `for` richiede tre variabili nella sua definizione:

```
 s h N {
i exch def
 istruzione 1
 istruzione 2
... ... ...
 istruzione k
} for
```

La variabile locale `i` parte con il valore pari ad `s` e viene incrementata di un valore pari ad `h` ad ogni esecuzione della procedura, fino a quando raggiunge il valore `N` e il ciclo termina.

Un modo simpatico di utilizzare questi cicli è il disegno di poligoni regolari; ad esempio la procedura:

```
1 0 moveto
1 1 5 { /i exch def
 i 72 mul cos i 72 mul sin lineto
} for
 closepath
```

disegna un pentagono regolare.

2.7 Istruzioni condizionali

In PostScript l'operatore `if` prende una variabile di tipo boolean ed un array eseguibile ed estrapola le operazioni contenute nell'array se il valore della variabile boolean è *true*.

Risulta utile quindi, aggiungere i comandi per confrontare due (o più) oggetti, definendo gli operatori:

`eq`	$=$	`ne`	$\neq$
`gt`	$>$	`lt`	$<$
`ge`	$\geq$	`le`	$\leq$

Questi operatori confrontano i primi due termini, che possono essere di qualsiasi tipo, dello stack e mandano nello stack una variabile di tipo boolean.

Capitolo 3

Disegnamo qualche frattale in PostScript

3.1 Il codice del Drago

Consideriamo il codice che genera la curva del dragone (*dragon.ps*).

Innanzitutto specifichiamo che si tratta di un file PostScript:

```
% !PS
```

A questo punto definiamo la variabile "order" che indica il numero di iterazione (è sconsigliabile superare le 10 iterazioni se si vuole che un algoritmo di medio bassa complessità riesca a completare la computazione).

```
/order 7 def
```

È il momento i scrivere l'algoritmo vero e proprio. Definiamo la parola di partenza ed il comportamento dei simboli X, Y, F, $+$ e -.

```
/START  X  def
```

Per quanto riguarda la X e la Y, definiamo due procedure che, se *order* non è nulla, ne riducono di 1 il valore e poi lo duplicano tante volte quante sono le chiamate di funzione che andranno effettuate (nel nostro caso 4). A questo punto vengono chiamate ricorsivamente le funzioni in base alla parola definita per l'algoritmo l-system preso in considerazione. Quando *order* è nulla la procedura si ferma.

```
/X {
  dup 0 ne
  {1 sub 4 {dup} repeat -  F X + + F Y -}
```

```
  if pop
} def

/Y {
  dup 0 ne
  {1 sub 4 {dup} repeat + F X - - F Y +}
  if pop
} def
```

La funzione F non fa nulla fino a che *order* non diventa 0. A quel punto disegna un segmento.

```
/F {
  0 eq { 10 0 rlineto } if
} bind def
```

Le funzioni + e - non sono altro che rotazioni di un certo angolo (nel nostro caso 45 gradi a destra e sinistra).

```
/- { -45 rotate } bind def
/+ { 45 rotate } bind def
```

Stabiliamo come debbano essere raccordati i segmenti:

```
1 setlinejoin
1 setlinecap
```

Infine stabiliamo dove far iniziare la stampa e quanto riscalare l'immagine in funzione dell'ordine, ruotiamo la nostra curva, eseguiamo la parola di avvio che avevamo memorizzato nella procedura *START* e rappresentiamo il risultato.

```
newpath
220 180 moveto
50 order { 2 sqrt div } repeat dup scale
90 rotate
order START
stroke

showpage
```

Avviando il codice appena descritto si otterrà la figura 3.1.

Mentre, eseguendo qualche iterazione in più (basta impostare *order* a 10), si ottiene la figura 3.2. Da notare come si vada raffinando la curva...

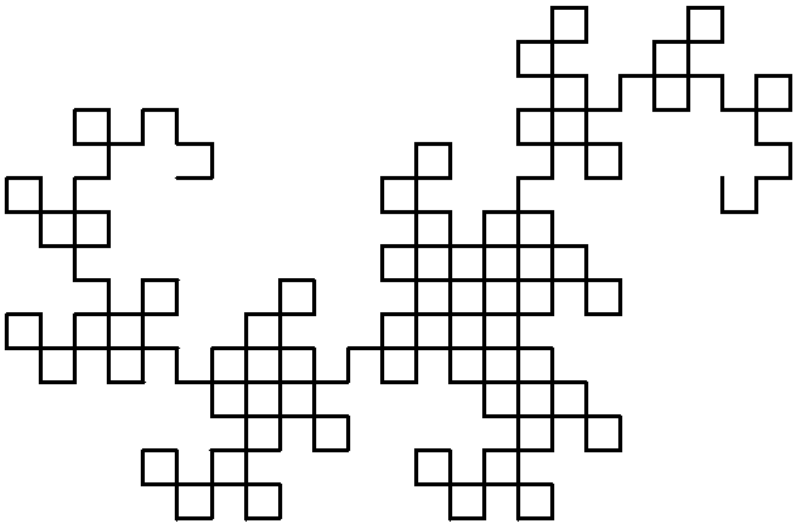

Figura 3.1: La curva del dragone

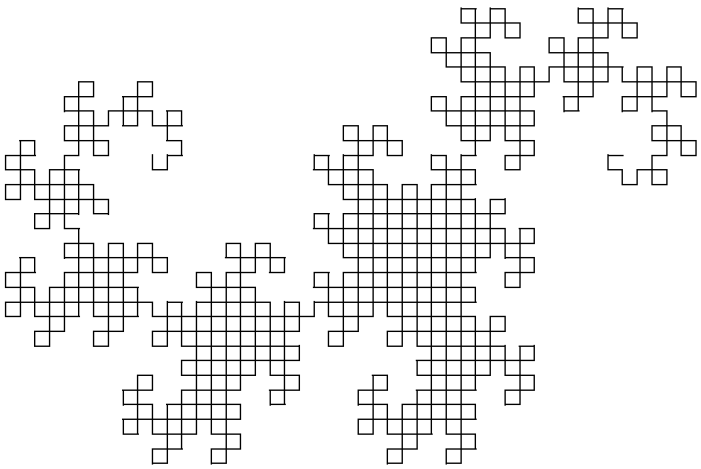

Figura 3.2: La curva del dragone

3.1.1 Semplici esperimenti sul codice di *dragon.ps*

Abbiamo provato a testare alcuni semplici algoritmi.

Programma curva_peano.ps

Il seguente codice

```
START = X
alpha = 90
X --> -YF+XFX+FY-
Y --> +XF-YFY-FX+
```

produce in output la figura 3.3

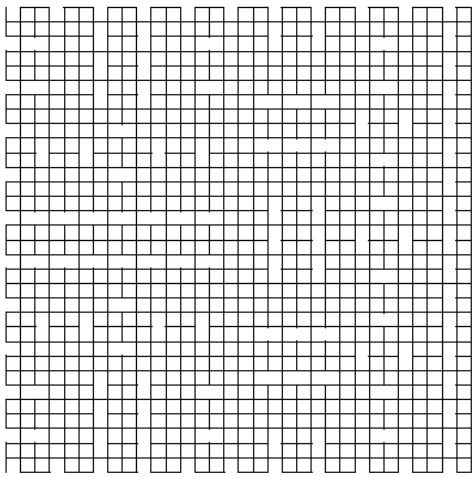

Figura 3.3: La curva di Peano

Programma koch.ps

Il seguente codice

```
START = X
alpha = 90
X --> +Y--Y--Y
Y --> YF+FY--YF+FY
```

produce in output la figura 3.4

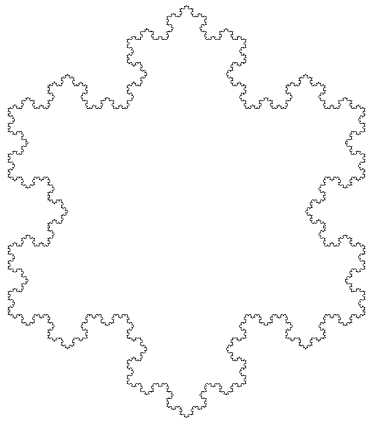

Figura 3.4: Il fiocco di neve di Koch

Programma variante_koch.ps

Il seguente codice

```
START = X
alpha = 60
X --> +YFY--YFY--YFY
Y --> YF+YFY--YFY+FY
```

produce in output la figura 3.5

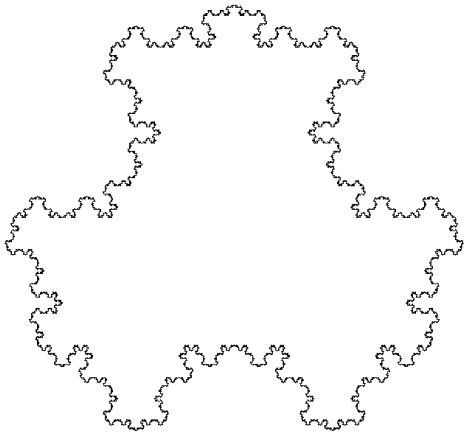

Figura 3.5: Un algoritmo che abbiamo ottenuto per caso

3.1.2 Semplici esperimenti sul codice di *ls-filo_erba.ps*

Abbiamo provato a testare alcuni semplici algoritmi. Si noti che questo codice non è stato commentato nel documento per la sua estrema complessità.

Programma ls_trombetta.ps

Il seguente codice

```
START = F
alpha = 30
F --> F[+F][-F]F
```

produce in output la figura 3.6

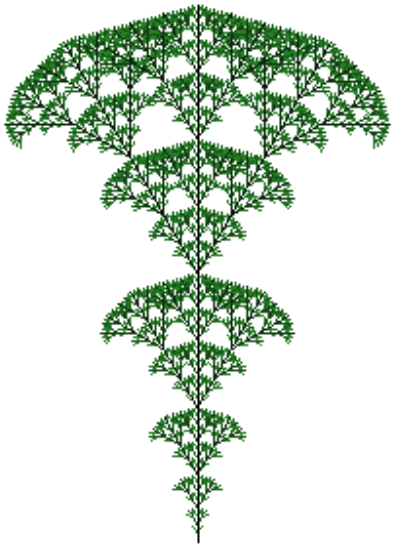

Figura 3.6: Un primo esempio di frattale nature-like

Programma ls_albero.ps

Il seguente codice

```
START = F
alpha = 22.5
F --> FF+[+F-F-F]-[-F+F+F]
```

produce in output la figura 3.7

Figura 3.7: Un cespuglio L-System

3.2 Le trasformazioni della felce ed altri suoi simili...

Riprendiamo in considerzione le successioni di insiemi piani di cui abbiamo ampiamente parlato prima. Indicheremo in ordine i coefficienti a, b, c, d, u, v delle contrazioni considerate.

Partiamo subito dall'analisi del codice (*felce.ps*) e dalla, ormai nota, indicazione del linguaggio usato:

```
%!PS-Adobe-1.0
```

Definiamo la serie dei coefficienti delle trasformazioni a, b, c, d, u, v:

```
/m1 [  0.00  0.00  0.00  0.16  0.00 0.00 ] def
/m2 [  0.85 -0.04  0.04  0.85  0.00 1.60 ] def
/m3 [  0.20  0.23 -0.26  0.22  0.00 1.60 ] def
/m4 [ -0.15  0.26  0.28  0.24  0.00 0.44 ] def
```

Definiamo alcune variabili utili

```
/point 72 def
/length 0.001 def
```

Facciamo in modo che la figura sia rappresentata in un quadrato di 8 pollici su una pagina di 8.5x11 pollici. Riscaliamo quindi tutto per fare in modo che la coordinata x vada da -5 a 5 e la coordinata y vada da 0 a 10. Infine impostiamo il primo punto all'origine.

```
4.25 point mul 1.5 point mul translate
0.8 point mul dup scale
1 setlinecap
0.005 setlinewidth
0 0
```

Il codice che segue, per 150.000 volte sceglie un valore casuale tra 0 e 100 ed applica allo spazio una delle quattro trasformazioni (scelta casualmente), dopodiché disegna un punto nella posizione in cui si trova.

```
150000 {
   /r rand 100 mod def
   r  1 lt { /m m1 def }
 { r 86 lt { /m m2 def }
 { r 93 lt { /m m3 def }
           { /m m4 def } ifelse } ifelse } ifelse
   m transform 2 copy moveto
   length length rlineto
   stroke
} repeat
```

Infine visualizziamo la pagina.

```
showpage
```

Mandando in esecuzione il codice si otterrà la figura 3.8

3.2.1 Semplici esperimenti sul codice di *felce.ps*

Abbiamo provato ad utilizzare vari semplici algoritmi per testare i risultati ottenuti.

Programma foglia.ps

Il seguente codice

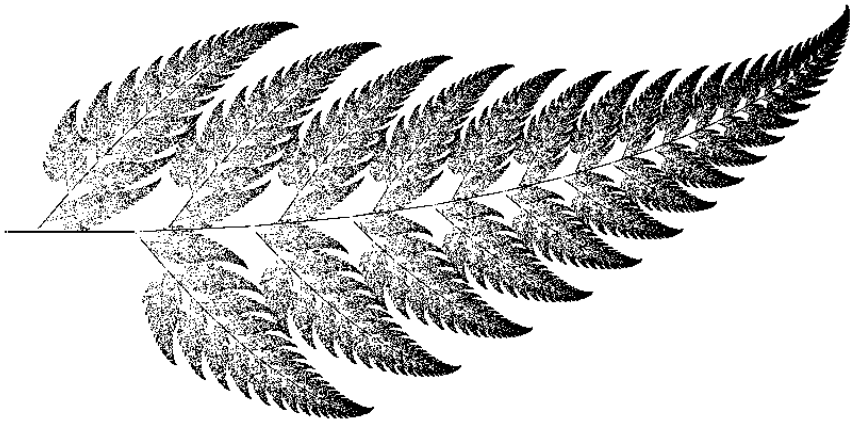

Figura 3.8: La Felce frattale

```
/m1 [  0.60  0.00  0.00  0.60  0.00 2.40 ] def
/m2 [  0.60  0.00  0.00  0.60  0.00 0.90 ] def
/m3 [  0.40 -0.30  0.30  0.40  0.00 1.80 ] def
/m4 [  0.40  0.30 -0.30  0.40  0.00 1.80 ] def
```

produce in output la figura 3.9

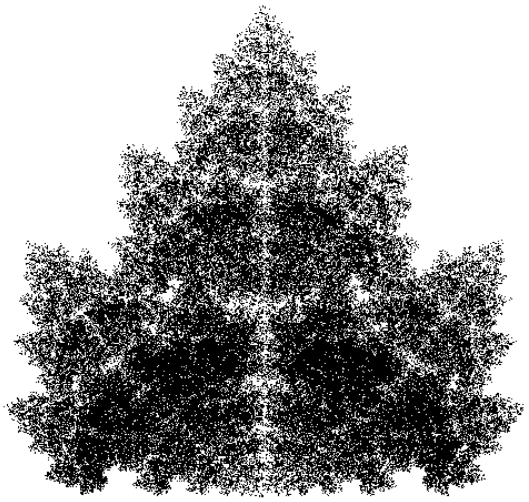

Figura 3.9: Una foglia frattale

Programma koch.ps

Il seguente codice

```
/m1 [  0.33  0.00  0.00  0.33  0.00 0.00 ] def
/m2 [  0.17  0.29 -0.29  0.17  0.33 0.00 ] def
/m3 [  0.17 -0.29  0.29  0.17  0.50 0.29 ] def
/m4 [  0.33  0.00  0.00  0.33  0.67 0.00 ] def
```

produce in output la figura 3.10

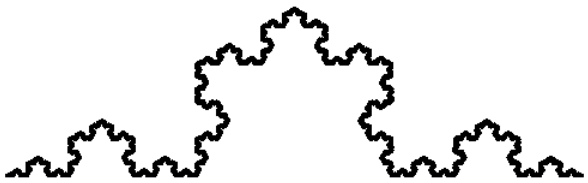

Figura 3.10: Un modo diverso di rappresentare il merletto di Koch

Programma sierpi.ps

Il seguente codice

```
/m1 [  0.50  0.00  0.00  0.50  0.00 0.00 ] def
/m2 [  0.50  0.00  0.00  0.50  0.50 0.00 ] def
/m3 [  0.50  0.00  0.00  0.50  0.00 0.50 ] def
```

produce in output la figura 3.11

cespuglio.ps

Il seguente codice

```
/m1 [  0.60  0.00  0.00  0.60  0.00 0.40 ] def
/m2 [  0.40  0.40 -0.20  0.50  0.00 0.20 ] def
/m3 [  0.40 -0.30  0.20  0.40  0.00 0.30 ] def
/m4 [  0.30  0.00  0.04  0.30  0.00 0.20 ] def
```

produce in output la figura 3.12

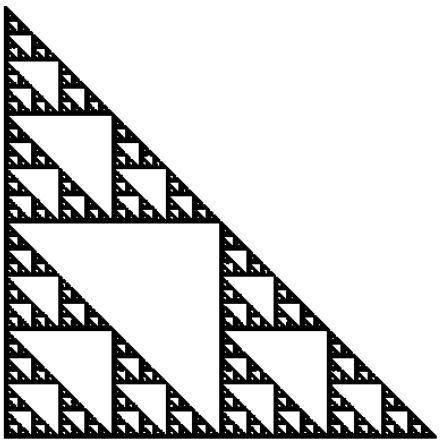

Figura 3.11: Il triangolo di Sierpinski

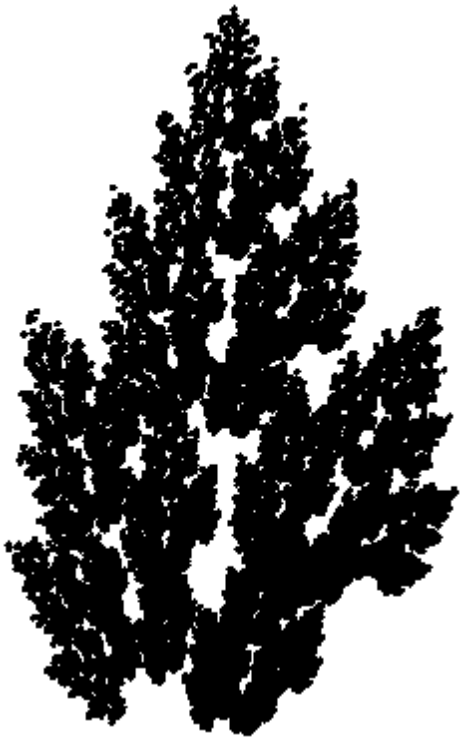

Figura 3.12: Un cespuglio frattale

Bibliografia

[Cas] Bill Casselman. Mathematical Illustrations - a manual of geometry and postscript (http://www.math.ubc.ca/~cass/graphics/manual/). PDF downloadable from the web.

[Fal03] Kenneth Falconer. *Fractal Geometry*. Wiley, 2003.

[LSP] L-Systems in PostScript (http://www.cs.unh.edu/~charpov/programming/l-systems/). Web site.

[PSF] PostScript Fractals (http://www.pvv.ntnu.no/~andersr/fractal/postscript.html). Web site.