



NoiStudiamo

DISPENSA DI LABORATORIO



Interazione e Multimedia

Laboratorio: grafica e interazione in Processing
e algoritmi di elaborazione delle immagini.

A CURA DI

Gabriele Fontanarosa

Ottobre 2026

Indice dei capitoli

Da Processing agli algoritmi di elaborazione delle immagini, con codice eseguito e verificato.

1. Introduzione a Processing	3
2. Primitive grafiche, modalità e colore	9
3. Interazione: mouse e tastiera	16
4. Trasformazioni affini	22
5. Classi, oggetti, ereditarietà e strutture dati	30
6. Immagini e pixel: PImage	37
7. Quantizzazione, interpolazione e PSNR	43
8. Operazioni puntuali, stretching ed equalizzazione	49
9. Rotazione, scaling e shear di immagini	55
10. Filtri di rango: mediano, massimo e minimo	62
11. Convoluzione: media N box, Sobel e Prewitt	66
12. Estrazione dei bit planes	71

Introduzione a Processing

Che cos'è Processing, come è fatto uno sketch, le funzioni `settings()`, `setup()` e `draw()`, il sistema di coordinate, variabili, funzioni matematiche e controllo di flusso.

Che cos'è Processing

Processing è un linguaggio di programmazione che permette di sviluppare giochi, animazioni e contenuti interattivi. **Eredita completamente la sintassi, i comandi e il paradigma a oggetti di Java** e in più mette a disposizione molte funzioni di alto livello per gestire con facilità la grafica e il multimedia. È distribuito con licenza libera GNU General Public License e funziona su Linux, macOS e Windows. In questa dispensa il codice è scritto nella modalità standard (Java) di Processing.

Un programma Processing si chiama **sketch** («schizzo») ed è salvato in un file con estensione `.pde` dentro una cartella con lo stesso nome. Gli eventuali file esterni (immagini, font, dati) vanno nella sottocartella `data` dello sketch: è lì che le funzioni di caricamento li cercano.

COME SONO STATI VERIFICATI GLI ESEMPI

Tutto il codice di questa dispensa è stato compilato ed eseguito con il motore grafico di Processing (la libreria core, in modalità Java), e le figure e i testi mostrati come «output» sono quelli prodotti davvero dall'esecuzione. Gli sketch che usano immagini lavorano sulle fotografie del corso (il fotografo e l'astronauta, 512 × 512 pixel), salvate nella cartella `data`.

Il primo sketch: modalità statica

Il programma più semplice è una sequenza di istruzioni: Processing le esegue una volta, dall'alto verso il basso, e mostra la finestra con il disegno risultante. È la **modalità statica**. Nell'esempio seguente (tratto dalle esercitazioni del corso) si disegnano tanti cerchi concentrici sempre più scuri verso il centro.

```
size(500, 500);
background(255);

int raggio = (int)(500 * sqrt(2));
noStroke();
for (int i = 0; i < raggio; i += 10) {
    int g = 255 - 255 * i / raggio;
    fill(g, g, g);
    ellipse(250, 250, raggio - i, raggio - i);
}
```

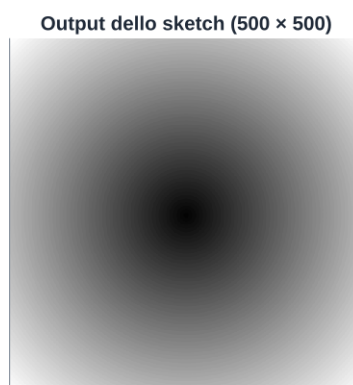


Figura 1. Cerchi concentrici con tonalità di grigio decrescente, disegnati dal più grande al più piccolo.

Si notano subito alcune regole che valgono sempre:

- `size(500, 500)` crea la finestra di disegno, larga 500 e alta 500 pixel; deve essere la prima istruzione. Dopo di essa le variabili `width` e `height` contengono le dimensioni.
- `background(255)` riempie lo sfondo con il grigio 255 (bianco). Un solo valore indica un livello di grigio da 0 (nero) a 255.
- `noStroke()` e `fill(g, g, g)` tolgono il contorno e impostano il colore di riempimento. Sono istruzioni di **stato**: restano valide per tutte le figure successive finché non vengono cambiate.
- Ogni istruzione termina con il punto e virgola; i blocchi sono racchiusi tra parentesi graffe; i commenti iniziano con `//`.

Modalità attiva: `setup()` e `draw()`

Per le animazioni e l'interazione si usa la **modalità attiva**, in cui il programma è organizzato in funzioni che Processing chiama al momento giusto. Quando nel codice compare almeno una tra `setup()` e `draw()`, le istruzioni non contenute in una funzione possono essere soltanto dichiarazioni di variabili e funzioni.

Funzione	Quando viene eseguita	A cosa serve
<code>settings()</code>	Una volta, prima di tutto	Imposta la dimensione della finestra quando questa dipende da variabili (<code>size()</code> con argomenti non costanti).
<code>setup()</code>	Una volta, all'avvio	Inizializzazioni: dimensione della finestra, frame rate, caricamento di immagini, stato iniziale.
<code>draw()</code>	Ripetutamente, per sempre	Disegna un fotogramma. Viene richiamata automaticamente <code>frameRate</code> volte al secondo.

Tabella 1. Le tre funzioni di struttura di uno sketch.

Il numero di fotogrammi al secondo si imposta con `frameRate(n)` (il valore predefinito è 60). Durante l'esecuzione la variabile `frameCount` conta i fotogrammi già disegnati, a partire da 1, e la variabile `frameRate` contiene la frequenza effettivamente raggiunta. Con `noLoop()` si ferma la chiamata automatica di `draw()` (se ne disegna un ultimo fotogramma), con `loop()` si riprende, con `redraw()` si richiede un solo fotogramma.

Esempio. Una pallina che rimbalza tra i bordi della finestra. La posizione `x` e la velocità `vx` sono dichiarate fuori dalle funzioni: sono variabili **globali** e mantengono il loro valore da un fotogramma al successivo. A ogni `draw()` la pallina avanza di `vx` pixel; quando supera un bordo il segno della velocità si inverte.

```

float x = 50;      // posizione orizzontale del centro
float vx = 6;      // velocità: pixel per fotogramma

void setup() {
  size(400, 160);
  frameRate(30);
}

void draw() {
  background(240);
  x = x + vx;
  if (x > width - 20 || x < 20) {
    vx = -vx;      // inverte il verso al bordo
  }
  fill(0, 71, 255);
  noStroke();
  ellipse(x, height / 2, 40, 40);
  println(frameCount, x);
}

```

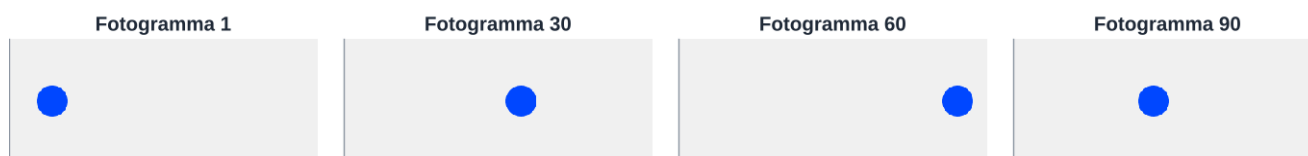


Figura 2. Quattro fotogrammi dello sketch del rimbalzo (400 × 160 pixel): la pallina parte da sinistra, rimbalza sul bordo destro e riparte.

A ogni fotogramma `println(frameCount, x)` scrive nella console il numero del fotogramma e la posizione. Le prime righe sono:

```

1 56.0
2 62.0
3 68.0
4 74.0

```

VARIABILI LOCALI E GLOBALI

Una variabile dichiarata **dentro** `draw()` esiste solo durante quel fotogramma: a ogni chiamata viene ricreata e riportata al valore iniziale. Una variabile dichiarata **fuori** dalle funzioni vive per tutta l'esecuzione. Lo sketch seguente mostra la differenza.

```

int globale = 0;      // dichiarata fuori da setup() e draw()

void setup() {
  size(100, 100);
}

void draw() {
  int locale = 0;      // ricreata a ogni fotogramma
  globale++;
  locale++;
  println(frameCount, globale, locale);
}

```

Dopo tre fotogrammi la console contiene (fotogramma, globale, locale):

```
1 1 1
2 2 1
3 3 1
```

SETTINGS() E DIMENSIONE DIPENDENTE DA VARIABILI

In `setup()` la dimensione della finestra si scrive nella forma `size(400, 300)` con numeri costanti: Processing la sposta automaticamente in `settings()`. Se invece le dimensioni sono calcolate (per esempio dal valore di una variabile) `size()` **non** può stare in `setup()` e si deve usare la funzione `settings()`, che viene eseguita per prima.

```
int lato = 240;

void settings() {
    size(lato, lato / 2);
}

void setup() {
    background(0, 71, 255);
    println(width, height);
    println(width - 1, height - 1);
}
```

La console mostra 240 120 e 239 119.

Il sistema di coordinate

Il punto (0, 0) è l'**angolo in alto a sinistra** della finestra. L'asse x cresce verso destra e l'asse y cresce **verso il basso**, al contrario del piano cartesiano usato in matematica. Una finestra larga `width` e alta `height` ha pixel con x da 0 a `width - 1` e y da 0 a `height - 1`: l'ultimo pixel non ha coordinata `width` ma `width - 1`.

Anche gli indici degli array, delle stringhe e dei vettori di pixel partono da 0 (come in tutto Java), non da 1: per un array con `n` elementi gli indici validi vanno da 0 a `n - 1`. È un fatto molto importante per gli algoritmi sulle immagini dei capitoli successivi.

```
int[] v = {5, 8, 13, 21};

void setup() {
    println("lunghezza:", v.length);
    println("primo:", v[0], "ultimo:", v[v.length - 1]);
    for (int i = 0; i < v.length; i++) {
        println(i, v[i]);
    }
}
```

```
lunghezza: 4
primo: 5 ultimo: 21
0 5
1 8
2 13
3 21
```

Variabili, tipi e operatori

Processing usa i tipi di Java. I principali sono:

Tipo	Contenuto	Esempio
int	numero intero (32 bit)	int n = 5;
float	numero con virgola (32 bit)	float a = 0.5;
boolean	vero o falso	boolean b = true;
char	un carattere	char c = 'a';
String	una stringa di testo	String s = "ciao";
color	un colore (capitolo 2)	color c = color(255, 0, 0);
int[]	array di interi	int[] v = new int[10];

Tabella 2. Tipi di dato di Processing.

I numeri decimali si scrivono con il punto (0.5), e in Processing sono automaticamente di tipo float. Gli operatori aritmetici sono +, -, *, / (resto della divisione) e le forme abbreviate +=, -=, *=, ++ e --. Due trabocchetti ricorrenti:

- la divisione tra due int è una **divisione intera** e scarta i decimali: 7 / 2 vale 3, mentre 7 / 2.0 vale 3,5;
- per convertire un float in int si usa il cast, (int) x, che **tronca** e non arrotonda ((int) 3.9 dà 3, round(3.9) dà 4).

Molte funzioni matematiche sono già disponibili. Gli angoli sono in **radianti**: le costanti PI, TWO_PI, HALF_PI e QUARTER_PI valgono π , 2π , $\pi/2$ e $\pi/4$, e le funzioni radians() e degrees() convertono tra gradi e radianti.

```
void setup() {  
  println(7 / 2, 7 / 2.0, 7 % 3);           // divisione intera, reale, resto  
  println((int) 3.9, round(3.5), floor(-1.2), ceil(-1.2));  
  println(constrain(15, 0, 10), map(5, 0, 10, 0, 100));  
  println(dist(0, 0, 3, 4), sqrt(16), pow(2, 10));  
  println(radians(180) == PI, degrees(HALF_PI));  
  println(min(4, 9, 2), max(4, 9, 2), abs(-7));  
}
```

L'output, riga per riga, è:

```
3 3.5 1  
3 4 -2 -1  
10 50.0  
5.0 4.0 1024.0  
true 90.0  
2 9 7
```

La funzione map() trasferisce un valore da un intervallo a un altro con una trasformazione lineare (qui da 0...10 a 0...100); constrain() limita un valore dentro un intervallo; dist() calcola la distanza euclidea tra due punti; random(n) restituisce un numero casuale con distribuzione uniforme in [0, n).

Controllo di flusso e funzioni

Le strutture di controllo sono quelle di Java: `if / else` per le scelte, `for` e `while` per i cicli, `switch` per le scelte multiple. Nei cicli annidati che scorrono su una griglia di pixel, il ciclo esterno è di solito quello sulle righe (`y`) e quello interno sulle colonne (`x`).

Una **funzione** si definisce indicando il tipo del risultato (`void` se non ne restituisce), il nome e i parametri. Nell'esempio, `dentro()` dice se un punto cade dentro un cerchio, e lo sketch colora di blu i quadratini di una griglia 20×20 il cui angolo cade dentro il cerchio di centro (100, 100) e raggio 80.

```
// restituisce true se il punto (x, y) cade dentro il cerchio di centro (cx, cy)
boolean dentro(float x, float y, float cx, float cy, float r) {
    return dist(x, y, cx, cy) < r;
}

void setup() {
    size(200, 200);
    background(255);
    int conta = 0;
    for (int y = 0; y < height; y += 10) {
        for (int x = 0; x < width; x += 10) {
            if (dentro(x, y, 100, 100, 80)) {
                fill(0, 71, 255);
                conta++;
            } else {
                fill(230);
            }
            rect(x, y, 8, 8);
        }
    }
    println("quadrati dentro il cerchio:", conta);
}
```

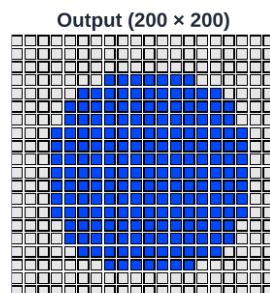


Figura 3. Griglia di quadratini colorati con la funzione `dentro()`: i quadrati blu sono quelli con l'angolo a distanza minore di 80 dal centro.

La console stampa `quadrati dentro il cerchio: 193`. Il valore è stato controllato contando a parte i punti (`x`, `y`), con `x` e `y` multipli di 10, a distanza minore di 80 dal centro: sono davvero 193. Ricorda che `print()` scrive sulla console senza andare a capo, `println()` va a capo; con più argomenti li separa con uno spazio: sono lo strumento principale di *debug*.

DA RICORDARE

Uno sketch ha una parte che gira una volta (`settings()`, `setup()`) e una che gira a ogni fotogramma (`draw()`). L'origine è in alto a sinistra e `y` cresce verso il basso; gli indici partono da 0; la divisione tra interi scarta i decimali; gli angoli sono in radianti. Le variabili globali mantengono il valore tra un fotogramma e l'altro, quelle locali no.

Primitive grafiche, modalità e colore

Le primitive `line()`, `rect()`, `ellipse()`, `bezier()` e `beginShape()`, i modificatori `rectMode()` e `ellipseMode()`, il tipo `color` e la funzione `color()`.

Le primitive principali

Una **primitiva** è una forma elementare che Processing sa disegnare con una sola chiamata. Tutte le coordinate sono in pixel, con origine in alto a sinistra. Ogni primitiva usa il colore di riempimento (`fill`) e il colore del contorno (`stroke`) correnti, come descritto più avanti in questo capitolo.

Funzione	Argomenti	Disegna
<code>point(x, y)</code>	posizione	un punto (un pixel con il colore del contorno)
<code>line(x1, y1, x2, y2)</code>	i due estremi	un segmento
<code>rect(x, y, w, h)</code>	angolo, larghezza, altezza	un rettangolo
<code>ellipse(x, y, w, h)</code>	centro, larghezza, altezza	un'ellisse (un cerchio se <code>w == h</code>)
<code>triangle(x1, y1, x2, y2, x3, y3)</code>	tre vertici	un triangolo
<code>quad(x1, y1, ..., x4, y4)</code>	quattro vertici	un quadrilatero
<code>arc(x, y, w, h, a1, a2)</code>	come ellipse, più due angoli	un arco, in senso orario (angoli in radianti)

Tabella 1. Le primitive bidimensionali di base.

```
size(600, 140);
background(255);
strokeWeight(2);
fill(0, 71, 255, 90);           // blu semitrasparente

line(20, 20, 100, 110);         // x1, y1, x2, y2
rect(120, 20, 80, 90);          // x, y, larghezza, altezza
ellipse(260, 65, 80, 60);       // centro x, centro y, larghezza, altezza
triangle(320, 110, 360, 20, 400, 110); // tre vertici
quad(430, 30, 500, 20, 510, 110, 440, 100); // quattro vertici
arc(570, 65, 50, 50, 0, PI + HALF_PI); // x, y, w, h, angolo iniziale, finale
```

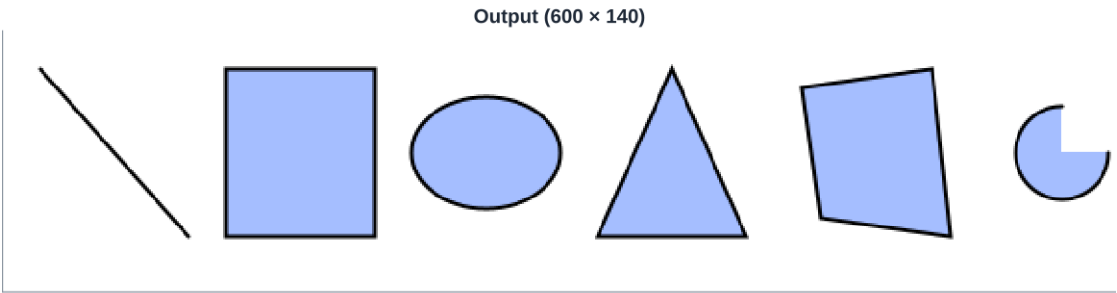


Figura 1. Dall'alto a sinistra: `line()`, `rect()`, `ellipse()`, `triangle()`, `quad()` e `arc()` con riempimento blu semitrasparente.

L'arco dell'esempio va da 0 a $\text{PI} + \text{HALF_PI}$: gli angoli si misurano a partire dall'asse x positivo (le ore 3) e crescono **in senso orario**, perché l'asse y punta verso il basso. Con la modalità predefinita (PIE) l'arco è chiuso con i due raggi; con OPEN o CHORD si ottengono un arco aperto o la sola corda.

Lo spessore del contorno si imposta con `strokeWeight(n)` (in pixel, predefinito 1). Con `noStroke()` il contorno scompare e con `noFill()` la figura resta senza riempimento.

rectMode(): come si interpretano gli argomenti di rect()

Per default i primi due argomenti di `rect()` sono le coordinate dell'angolo in alto a sinistra. La funzione `rectMode(mode)` cambia il significato di tutti e quattro gli argomenti, e la scelta resta valida per i rettangoli successivi finché non si chiama di nuovo `rectMode()`.

Modalità	Argomenti di rect(a, b, c, d)
CORNER	(a, b) è l'angolo in alto a sinistra, c è la larghezza, d è l'altezza (predefinita)
CENTER	(a, b) è il centro, c è la larghezza, d è l'altezza
CORNERS	(a, b) e (c, d) sono due angoli opposti
RADIUS	(a, b) è il centro, c e d sono i semiassi (metà di larghezza e altezza)

Tabella 2. Le quattro modalità di `rectMode()`.

```
size(600, 140);
background(255);
fill(0, 71, 255, 100);
stroke(0);

rectMode(CORNER);    // predefinito: (x, y) è l'angolo in alto a sinistra
rect(30, 30, 100, 80);

rectMode(CENTER);    // (x, y) è il centro
rect(210, 70, 100, 80);

rectMode(CORNERS);   // (x1, y1) e (x2, y2) sono due angoli opposti
rect(320, 30, 420, 110);

rectMode(RADIUS);    // (x, y) è il centro; gli ultimi due sono i semiassi
rect(510, 70, 50, 40);
```

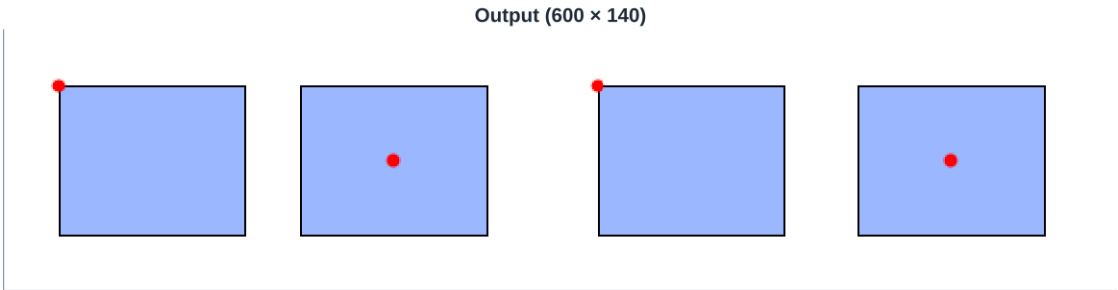


Figura 2. Lo stesso rettangolo di 100 × 80 pixel disegnato con `CORNER`, `CENTER`, `CORNERS` e `RADIUS`; il punto rosso (disegnato da poche righe omesse nel listato) è la posizione indicata dai primi due argomenti.

I quattro rettangoli hanno tutti le stesse dimensioni, 100 × 80 pixel: cambia soltanto il modo in cui sono stati descritti. Per esempio, in modalità `CORNERS` gli argomenti 320, 30, 420, 110 significano «due angoli opposti», quindi larghezza e altezza sono $420 - 320 = 100$ e $110 - 30 = 80$.

ellipseMode(): lo stesso discorso per le ellissi

Per le ellissi vale lo stesso schema, ma il valore predefinito è CENTER (centro, larghezza, altezza), cioè quello che per i rettangoli è CENTER. La funzione `ellipseMode(mode)` accetta gli stessi quattro valori.

Modalità	Argomenti di <code>ellipse(a, b, c, d)</code>
CENTER	(a, b) è il centro, c è la larghezza, d è l'altezza (predefinita)
RADIUS	(a, b) è il centro, c e d sono i raggi orizzontale e verticale
CORNER	(a, b) è l'angolo in alto a sinistra del rettangolo che contiene l'ellisse
CORNERS	(a, b) e (c, d) sono due angoli opposti di quel rettangolo

Tabella 3. Le quattro modalità di `ellipseMode()`.

```
size(600, 140);
background(255);
fill(255, 140, 0, 110);
stroke(0);

ellipseMode(CENTER);    // predefinito: centro, larghezza, altezza
ellipse(70, 70, 100, 60);

ellipseMode(RADIUS);    // centro, raggio orizzontale, raggio verticale
ellipse(210, 70, 50, 30);

ellipseMode(CORNER);    // angolo in alto a sinistra del rettangolo, larghezza, altezza
ellipse(330, 40, 100, 60);

ellipseMode(CORNERS);   // due angoli opposti del rettangolo
ellipse(470, 40, 570, 100);
```

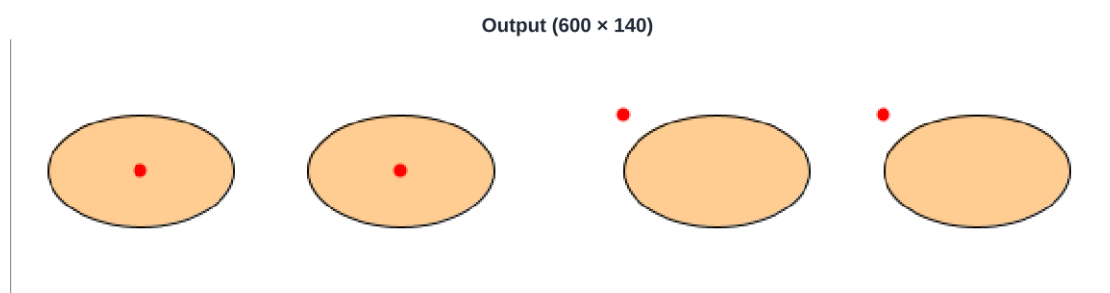


Figura 3. La stessa ellisse di 100 × 60 pixel descritta in quattro modi diversi; il punto rosso (disegnato da righe omesse nel listato) è il primo argomento.

ATTENZIONE

Le modalità sono uno **stato globale**. Se si cambia `rectMode()` in un punto del programma e si dimentica di ripristinarlo, tutte le figure successive vengono interpretate in modo diverso. Dentro `draw()` conviene richiamare le modalità che servono a ogni fotogramma, oppure lasciarle impostate una volta sola in `setup()`.

Curve di Bézier: bezier()

La funzione `bezier(x1, y1, cx1, cy1, cx2, cy2, x2, y2)` disegna una **curva di Bézier cubica**. Gli otto argomenti sono le coordinate di quattro punti: il punto iniziale P_0 , i due **punti di controllo** C_1 e C_2 e il punto finale P_1 . La curva parte da P_0 in direzione di C_1 e arriva in P_1 provenendo dalla direzione di C_2 , ma **non passa** per i punti di controllo: essi la «attirano» soltanto.

$$P(t) = (1 - t)^3 P_0 + 3(1 - t)^2 t C_1 + 3(1 - t) t^2 C_2 + t^3 P_1 \quad \text{con } t \text{ tra } 0 \text{ e } 1$$

Per $t = 0$ si ottiene P_0 e per $t = 1$ si ottiene P_1 . Per $t = 1/2$ la formula dà $P = (P_0 + 3C_1 + 3C_2 + P_1) / 8$: nell'esempio, con $P_0 = (40, 160)$, $C_1 = (100, 20)$, $C_2 = (300, 20)$ e $P_1 = (360, 160)$, il punto centrale della curva è $(200, 55)$.

```
size(400, 200);
background(255);
noFill();

stroke(0, 71, 255);
strokeWeight(3);
bezier(40, 160, 100, 20, 300, 20, 360, 160);    // P0, C1, C2, P1

stroke(255, 0, 0);                               // segmenti e punti di controllo
strokeWeight(1);
line(40, 160, 100, 20);
line(360, 160, 300, 20);
fill(255, 0, 0);
ellipse(100, 20, 8, 8);
ellipse(300, 20, 8, 8);
```

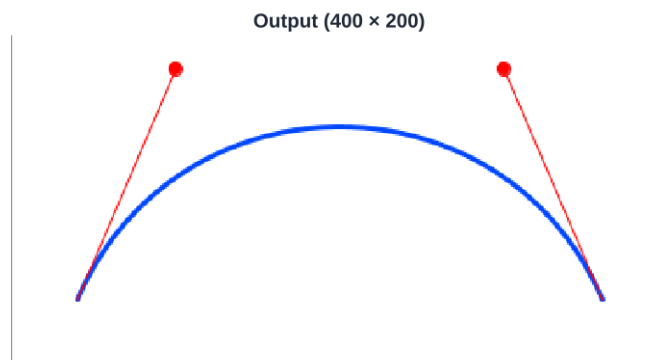


Figura 4. Curva di Bézier cubica (blu) con i segmenti e i punti di controllo (rossi).

Il calcolo è stato verificato sull'immagine prodotta dallo sketch: sulla colonna $x = 200$ i pixel blu del tratto (spesso 3 pixel) occupano le righe da 54 a 56, il cui centro è proprio $y = 55$. I segmenti da P_0 a C_1 e da P_1 a C_2 sono le **tangenti** alla curva agli estremi: è il modo con cui si controlla la forma.

Forme libere: beginShape(), vertex(), endShape()

Per disegnare poligoni qualsiasi si elencano i vertici uno dopo l'altro tra `beginShape()` e `endShape()`. Per default la figura è **aperta** (l'ultimo lato non viene tracciato, anche se il riempimento la chiude comunque); passando `CLOSE` a `endShape()` si chiude anche il contorno.

Come argomento di `beginShape()` si può indicare un tipo di figura, che dice come interpretare la sequenza di vertici: `POINTS` (punti separati), `LINES` (segmenti a coppie), `TRIANGLES` (triangoli a terne), `TRIANGLE_STRIP` (ogni vertice forma un triangolo con i due precedenti), `TRIANGLE_FAN` (ventaglio attorno al primo vertice), `QUADS` e `QUAD_STRIP` per i quadrilateri.

```

size(600, 160);
background(255);
fill(0, 71, 255, 90);
stroke(0);
strokeWeight(2);

beginShape();                // aperta: l'ultimo lato non viene chiuso
vertex(20, 120); vertex(60, 20); vertex(100, 120); vertex(140, 20);
endShape();

beginShape();                // chiusa con endShape(CLOSE)
vertex(180, 120); vertex(220, 20); vertex(260, 120); vertex(300, 20);
endShape(CLOSE);

beginShape();                // stella a cinque punte con un ciclo
for (int i = 0; i < 10; i++) {
    float r = (i % 2 == 0) ? 60 : 25;
    float a = -HALF_PI + i * PI / 5;
    vertex(400 + r * cos(a), 80 + r * sin(a));
}
endShape(CLOSE);

beginShape(TRIANGLE_STRIP);  // striscia di triangoli
vertex(480, 130); vertex(500, 30); vertex(520, 130);
vertex(540, 30); vertex(560, 130); vertex(580, 30);
endShape();

```

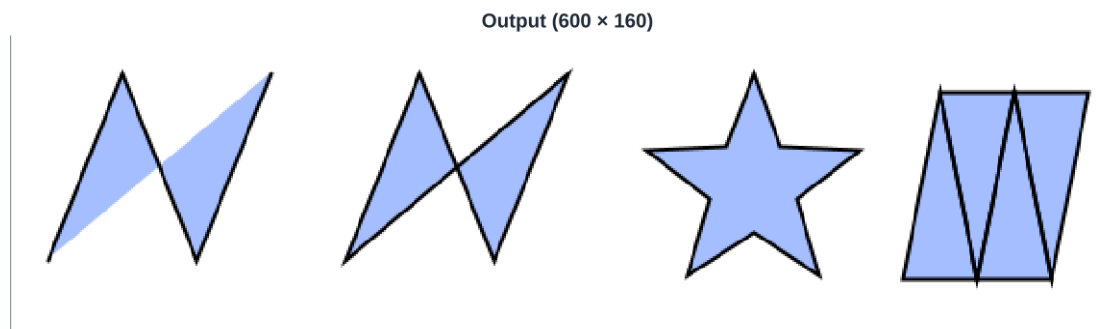


Figura 5. Da sinistra: forma aperta, forma chiusa con `CLOSE`, stella a cinque punte costruita con un ciclo e striscia di triangoli.

La stella è disegnata con dieci vertici su due circonferenze: i vertici di indice pari stanno sul raggio 60 (punte), quelli di indice dispari sul raggio 25 (rientranze), ogni 36° ($\pi/5$ radianti) a partire dall'alto (angolo `-HALF_PI`). Le coordinate di ciascun vertice sono $(x_c + r \cos a, y_c + r \sin a)$. Nella striscia di triangoli i sei vertici formano quattro triangoli: il primo con i vertici 1, 2, 3, il secondo con 2, 3, 4 e così via.

Il colore: tipo `color` e `color()`

Il colore si imposta con `fill()` (riempimento), `stroke()` (contorno) e `background()` (sfondo); ciascuna accetta gli stessi argomenti:

Chiamata	Significato
<code>fill(g)</code>	livello di grigio da 0 (nero) a 255 (bianco)
<code>fill(g, a)</code>	grigio con trasparenza a (0 trasparente, 255 opaco)
<code>fill(r, g, b)</code>	componenti rosso, verde e blu, ciascuna da 0 a 255
<code>fill(r, g, b, a)</code>	come sopra, con trasparenza
<code>fill(#0047FF)</code>	colore esadecimale RRGGBB, come nei programmi di grafica
<code>fill(c)</code>	un valore di tipo <code>color</code> già costruito

Tabella 4. Le forme di `fill()`, `stroke()` e `background()`.

Il **tipo** `color` contiene un colore in un solo numero intero a 32 bit, in formato **ARGB**: 8 bit per la trasparenza (alpha), 8 per il rosso, 8 per il verde, 8 per il blu. La funzione `color(r, g, b)` costruisce questo valore da tre componenti; per esempio il rosso puro è `color(255, 0, 0)`, cioè in esadecimale `FFFF0000`. Siccome il bit più alto vale 1, il numero intero risulta **negativo**: per i colori opachi il valore è sempre negativo.

```
color rosso, arancio, grigio, velato, mix, hsb;

void setup() {
  rosso = color(255, 0, 0);
  arancio = #FF8800;
  grigio = color(128);
  velato = color(0, 71, 255, 128);
  println(rozzo, hex(rozzo));
  println(red(rozzo), green(rozzo), blue(rozzo), alpha(rozzo));
  println(hex(arancio), hex(grigio), hex(velato));
  println(red(velato), green(velato), blue(velato), alpha(velato));
  println((arancio >> 16) & 0xFF, (arancio >> 8) & 0xFF, arancio & 0xFF);
  mix = lerpColor(color(255, 0, 0), color(0, 0, 255), 0.5);
  println(hex(mix));
  colorMode(HSB, 360, 100, 100);
  hsb = color(30, 100, 100);
  println(hue(hsb), saturation(hsb), brightness(hsb));
  colorMode(RGB, 255);
  println(red(hsb), green(hsb), blue(hsb));
}
```

Riga per riga, la console mostra:

```
-65536 FFFF0000
255.0 0.0 0.0 255.0
FFFF8800 FF808080 800047FF
0.0 71.0 255.0 128.0
255 136 0
FF800080
29.88235 100.0 100.0
255.0 127.0 0.0
```

- `-65536, FFFF0000`: il rosso puro come intero (`-65536 = 0xFFFF0000`) e in esadecimale; `hex(c)` ne scrive la forma ARGB in 8 cifre.
- `red()`, `green()`, `blue()` e `alpha()` estraggono le componenti dal colore e le restituiscono come `float` (anche la trasparenza: 255,0 per un colore opaco);

- un grigio `color(128)` ha le tre componenti uguali (808080) e un colore con trasparenza 128 ha alpha 80 in esadecimale (80 sono i due bit in testa);
- le componenti si possono anche ricavare con operazioni sui bit, $(c \gg 16) \& 0xFF$ per il rosso, $(c \gg 8) \& 0xFF$ per il verde e $c \& 0xFF$ per il blu, come nella quinta riga (255 136 0 per #FF8800);
- `lerpColor(c1, c2, t)` calcola il colore a metà strada tra due colori, componente per componente: tra rosso e blu dà 800080 (viola, con 127,5 arrotondato a 128).

COLORMODE(): LAVORARE IN HSB

Per default i colori sono interpretati come RGB con componenti da 0 a 255. Con `colorMode(HSB, 360, 100, 100)` si passa al modello **HSB** (tonalità, saturazione, luminosità) e si scelgono i fondi scala: nell'esempio *H* da 0 a 360 e *S, B* da 0 a 100. Il colore `color(30, 100, 100)` è un arancione pieno. Ripristinando `colorMode(RGB, 255)` e rileggendo le componenti si ottiene 255 127 0, cioè (255, 127, 0): la conversione tronca 127,5 a 127, ed è per questo che rileggendo la tonalità si trova 29.88 e non esattamente 30. Il modello HSB è descritto nel capitolo 16 della dispensa di teoria.

DA RICORDARE

Le primitive usano lo **stato** corrente: riempimento, contorno, spessore, modalità. `rectMode()` e `ellipseMode()` cambiano il significato degli argomenti di `rect()` e `ellipse()` (per questa il predefinito è CENTER). Una curva di Bézier cubica passa per il primo e l'ultimo punto ma non per quelli di controllo. Il tipo `color` è un intero ARGB a 32 bit.

Interazione: mouse e tastiera

Le variabili `mouseX`, `mouseY`, `mouseButton`, `key` e `keyCode` e le funzioni di evento `mousePressed()`, `mouseReleased()`, `mouseClicked()`, `mouseDragged()`, `mouseMoved()` e `keyPressed()`.

Eventi e funzioni di callback

Un programma interattivo reagisce a ciò che fa l'utente. In Processing questo avviene in due modi complementari:

- **lettura di variabili** (polling): dentro `draw()` si leggono i valori che Processing aggiorna da solo, come `mouseX` e `mousePressed`. Si vede lo stato attuale, a ogni fotogramma;
- **funzioni di evento** (callback): si scrivono funzioni con un nome prefissato, come `mousePressed()`, che Processing chiama **una volta** nel momento esatto in cui l'evento accade.

Le funzioni di evento non si chiamano mai direttamente: basta definirle. Se non servono si omettono. Per ricevere gli eventi lo sketch deve essere in modalità attiva (con `setup()` o `draw()`, anche vuota).

COME SONO STATI VERIFICATI GLI ESEMPI

Una finestra senza utente non riceve eventi veri, quindi i test di questo capitolo sono stati eseguiti riproducendo gli eventi con uno script (posizione del mouse, tasto premuto, rilascio, trascinamento, tasto della tastiera) che richiama le funzioni dello sketch nell'ordine in cui le chiamerebbe Processing. Le figure e gli output mostrati sono quelli che ne risultano.

Il mouse: variabili

Variabile	Tipo	Contenuto
<code>mouseX</code>	<code>int</code>	coordinata orizzontale del cursore, in pixel
<code>mouseY</code>	<code>int</code>	coordinata verticale del cursore, in pixel
<code>pmouseX</code>	<code>int</code>	coordinata orizzontale precedente (posizione al fotogramma o all'evento prima)
<code>pmouseY</code>	<code>int</code>	coordinata verticale precedente
<code>mousePressed</code>	<code>boolean</code>	vero finché un tasto del mouse è premuto
<code>mouseButton</code>	<code>int</code>	tasto premuto: LEFT, RIGHT oppure CENTER

Tabella 1. Le variabili del mouse.

La posizione è espressa nello stesso sistema di coordinate della finestra, quindi `mouseX` e `mouseY` possono essere usate direttamente come argomenti di `ellipse()`, `line()` e delle altre primitive. Il segmento `line(pmouseX, pmouseY, mouseX, mouseY)` unisce la posizione precedente a quella attuale: è la base di ogni programma di disegno a mano libera.

Il mouse: funzioni di evento

Funzione	Quando viene chiamata
<code>mousePressed()</code>	una volta, quando si preme un tasto del mouse
<code>mouseReleased()</code>	una volta, quando il tasto viene rilasciato
<code>mouseClicked()</code>	dopo il rilascio, se il mouse è rimasto fermo tra pressione e rilascio (un clic completo)
<code>mouseDragged()</code>	a ogni movimento del mouse con un tasto premuto
<code>mouseMoved()</code>	a ogni movimento del mouse con nessun tasto premuto

Tabella 2. Le funzioni di evento del mouse.

Lo sketch seguente non disegna nulla: stampa nella console ciò che accade. Come risposta a una sequenza di eventi simulati (due spostamenti, un clic con il tasto sinistro, poi pressione con il tasto destro, due trascinamenti e rilascio) si ottiene l'output riportato sotto.

```
void setup() {
  size(200, 100);
}

void draw() {
}

void mousePressed() {
  println("mousePressed", mouseX, mouseY, mouseButton == LEFT ? "LEFT" : "RIGHT");
}

void mouseReleased() {
  println("mouseReleased", mouseX, mouseY);
}

void mouseClicked() {
  println("mouseClicked", mouseX, mouseY);
}

void mouseDragged() {
  println("mouseDragged", mouseX, mouseY, pmouseX, pmouseY);
}

void mouseMoved() {
  println("mouseMoved", mouseX, mouseY);
}
```

```
mouseMoved 20 30
mouseMoved 40 30
mousePressed 40 30 LEFT
mouseReleased 40 30
mouseClicked 40 30
mousePressed 60 50 RIGHT
mouseDragged 90 70 60 50
mouseDragged 120 70 90 70
mouseReleased 120 70
```

Si osservano le regole principali:

- un clic completo genera, in ordine, `mousePressed()`, `mouseReleased()` e `mouseClicked()`: nello stesso punto e per lo stesso tasto;
- se il mouse si sposta tra pressione e rilascio `mouseClicked()` non viene chiamata (ultimi eventi: pressione, trascinamenti, rilascio); `mouseDragged()` e `mouseMoved()` sono distinte e si escludono a vicenda;
- dentro `mouseDragged()` le variabili `mouseX` e `mouseY` contengono la posizione attuale e quella dell'evento precedente (nel primo trascinamento 60 50, il punto della pressione);
- `mouseButton` vale `LEFT` o `RIGHT` a seconda del tasto: si distinguono così le azioni del tasto sinistro e del destro.

VARIABILE MOUSEPRESSED E FUNZIONE MOUSEPRESSED()

Il nome `mousePressed` indica due cose diverse. La **variabile** booleana è vera per tutto il tempo in cui il tasto è premuto, e si legge in `draw()` (di solito per animazioni continue). La **funzione** viene eseguita una sola volta per ogni pressione, ed è adatta ad azioni puntuali, come aprire un menu o cambiare stato. Lo stesso vale per `keyPressed` e la funzione `keyPressed()` per la tastiera.

L'esempio seguente è un **pulsante** a due stati. La funzione `sopra()` verifica se il cursore è sopra il rettangolo e in tal caso inverte la variabile `acceso`; la variabile `mousePressed` serve invece a colorare di blu il pulsante finché è premuto.

```
boolean acceso = false;

void setup() {
  size(240, 120);
}

void draw() {
  background(acceso ? color(255, 215, 0) : color(60));
  if (mousePressed && sopra()) fill(0, 71, 255); // mentre è premuto
  else fill(255);
  stroke(0);
  rect(80, 40, 80, 40, 8);
}

boolean sopra() {
  return mouseX > 80 && mouseX < 160 && mouseY > 40 && mouseY < 80;
}

void mousePressed() {
  if (sopra()) {
    acceso = !acceso;
    println("acceso =", acceso);
  }
}
```



Figura 1. Il pulsante: stato iniziale, pressione, rilascio (acceso), clic fuori dal pulsante (nessun effetto) e secondo clic sul pulsante (spento).

Nella console compaiono soltanto le due righe prodotte dai clic sul pulsante, mentre il clic fuori dal rettangolo non produce nulla:

```
acceso = true  
acceso = false
```

Disegnare con il mouse e la tastiera

Mettendo insieme eventi del mouse e della tastiera si costruisce un piccolo programma di disegno: il trascinamento con il tasto sinistro traccia una linea, con il tasto destro cancella (disegna in bianco con un tratto più largo), e alcuni tasti cambiano colore e spessore o puliscono la tela.

```
float spessore = 3;  
color tratto;  
  
void setup() {  
  size(400, 200);  
  background(255);  
  tratto = color(0, 71, 255);  
}  
  
void draw() {  
}  
  
void mouseDragged() {  
  if (mouseButton == RIGHT) {  
    stroke(255);           // tasto destro: gomma  
    strokeWeight(5 * spessore);  
  } else {  
    stroke(tratto);        // tasto sinistro: penna  
    strokeWeight(spessore);  
  }  
  line(pmouseX, pmouseY, mouseX, mouseY);  
}  
  
void keyPressed() {  
  if (key == 'c') background(255);  
  else if (key == 'r') tratto = color(255, 0, 0);  
  else if (key == 'b') tratto = color(0, 71, 255);  
  else if (key == '+') spessore++;  
  else if (key == '-') spessore = max(1, spessore - 1);  
}
```

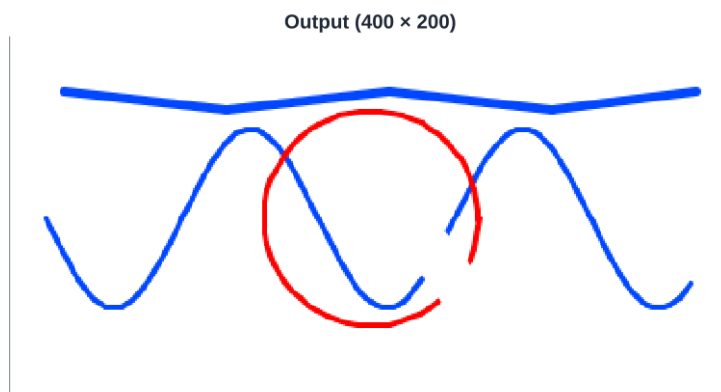


Figura 2. Risultato di una sessione simulata: una sinusoide blu, un cerchio rosso (tasto r), una linea più spessa blu (tasto + due volte, poi b) e un tratto di gomma con il tasto destro.

Il codice usa lo schema tipico: lo **stato** (colore e spessore) sta in variabili globali, `keyPressed()` lo modifica, `mouseDragged()` lo usa. Non c'è nulla in `draw()` perché il disegno si accumula sulla finestra senza essere cancellato: dopo `setup()` i vecchi tratti restano visibili, e per questo `background(255)` viene chiamato solo se si preme `c`.

La tastiera

Elemento	Contenuto
<code>key</code>	il carattere dell'ultimo tasto premuto (tipo <code>char</code>); distingue maiuscole e minuscole
<code>keyCode</code>	codice numerico del tasto; serve soprattutto per i tasti speciali
<code>keyPressed</code>	variabile booleana: vera finché un tasto è premuto
<code>keyPressed()</code>	funzione chiamata a ogni pressione di un tasto
<code>keyReleased()</code>	funzione chiamata al rilascio del tasto
<code>keyTyped()</code>	funzione chiamata per i tasti che producono un carattere

Tabella 3. Variabili e funzioni della tastiera.

Per i tasti normali si confronta direttamente `key` con un carattere tra apici, per esempio `key == 'a'`. Per i **tasti speciali** (le frecce, Maiusc, Ctrl, Alt) `key` non contiene un carattere stampabile e vale la costante `CODED`: in quel caso si legge `keyCode` e lo si confronta con `UP`, `DOWN`, `LEFT` e `RIGHT`. Il controllo `key == CODED` va fatto per primo.

```
void setup() {
  println(UP, DOWN, LEFT, RIGHT);
  println(CODED, int(ENTER), int(RETURN), int(ESC));
  println(int(BACKSPACE), int(TAB), int(DELETE));
}

void keyPressed() {
  if (key == CODED) {
    println("speciale: keyCode", keyCode);
  } else {
    println("carattere:", key, "int", int(key), "keyCode", keyCode);
  }
}
```

La console, dopo l'avvio e la pressione dei tasti `a`, `Maiusc` più `a`, `5`, `barra spaziatrice`, `freccia su` e `freccia sinistra`, mostra:

```
38 40 37 39
65535 10 13 27
8 9 127
carattere: a int 97 keyCode 65
carattere: A int 65 keyCode 65
carattere: 5 int 53 keyCode 53
carattere:  int 32 keyCode 32
speciale: keyCode 38
speciale: keyCode 37
```

- la prima riga contiene i valori di `UP`, `DOWN`, `LEFT` e `RIGHT` (38, 40, 37 e 39), la seconda e la terza quelli di `CODED` e dei tasti `Invio`, `Ritorno`, `Esc`, `Backspace`, `Tab` e `Canc` (10, 13, 27, 8, 9, 127): sono i codici ASCII di quei caratteri;

- per le lettere key ha il valore del carattere ASCII **minuscolo** (a vale 97, A vale 65), mentre keyCode vale sempre il codice della **maiuscola** (65 per entrambe): per distinguere le due versioni serve key;
- per le cifre e per la barra spaziatrice i due valori coincidono (53 e 32);
- per i tasti speciali compare soltanto keyCode.

Il tasto Esc chiude lo sketch; per intercettarlo senza uscire si imposta `key = 0` alla fine di `keyPressed()`. Se si tiene premuto un tasto, il sistema operativo ripete la pressione e `keyPressed()` viene chiamata più volte.

L'esempio seguente muove un quadrato di 20 pixel con le frecce, a passi di 20 pixel; la funzione `constrain()` impedisce che esca dalla finestra (il centro resta tra 10 e `width - 10`).

```
float x = 100, y = 100;

void setup() {
  size(200, 200);
}

void draw() {
  background(255);
  fill(0, 71, 255);
  rect(x - 10, y - 10, 20, 20);
}

void keyPressed() {
  if (key == CODED) {
    if (keyCode == LEFT) x -= 20;
    if (keyCode == RIGHT) x += 20;
    if (keyCode == UP) y -= 20;
    if (keyCode == DOWN) y += 20;
  }
  x = constrain(x, 10, width - 10); // il quadrato non esce dalla finestra
  y = constrain(y, 10, height - 10);
  println(x, y);
}
```

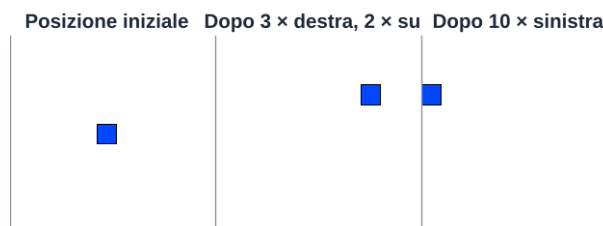


Figura 3. Il quadrato mosso con le frecce: nell'ultima figura il vincolo di `constrain()` lo ferma a `x = 10`.

Le coordinate stampate cominciano con `120.0 100.0` (centro del quadrato alla partenza, una pressione di freccia destra) e dopo le dieci pressioni di freccia sinistra finiscono con `10.0 60.0`: le ultime due non cambiano nulla perché il vincolo ha già fermato il quadrato. È il tipico uso di `constrain()` per tenere un oggetto dentro la finestra.

DA RICORDARE

Le variabili (`mouseX`, `mouseY`, `mousePressed`) dicono *com'è* il mouse adesso; le funzioni di evento (`mousePressed()`, `mouseReleased()`, `mouseClicked()`, `mouseDragged()`) vengono chiamate *quando succede* qualcosa. Un clic completo è pressione, rilascio e clic nello stesso punto. Per i tasti normali si usa `key`, per le frecce `keyCode` dopo il controllo di `key == CODED`. Gli stati dello sketch (modalità, colori, interruttori) vanno tenuti in variabili globali.

Trasformazioni affini

Le funzioni `translate()`, `rotate()`, `scale()`, `shearX()`, `shearY()`, `applyMatrix()`, `pushMatrix()` e `popMatrix()` e il loro legame con le matrici affini della teoria.

Muovere il sistema di coordinate

In Processing una trasformazione non sposta le figure già disegnate: modifica il **sistema di coordinate** con cui vengono interpretate le istruzioni **successive**. Internamente Processing tiene una matrice 3×3 (in coordinate omogenee, di cui mostra le prime due righe) che trasforma ogni punto prima di disegnarlo: ogni chiamata a una funzione di trasformazione la moltiplica per una nuova matrice, e l'effetto si accumula. All'inizio di ogni `draw()` la matrice torna all'identità.

Funzione	Effetto
<code>translate(tx, ty)</code>	sposta l'origine di tx a destra e di ty in basso
<code>rotate(a)</code>	ruota gli assi attorno all'origine dell'angolo a, in radianti , in senso orario sullo schermo
<code>scale(s)</code> / <code>scale(sx, sy)</code>	moltiplica le coordinate per s, oppure per sx sull'asse x e sy sull'asse y
<code>shearX(a)</code>	inclina: ogni punto si sposta in orizzontale di $y \cdot \tan(a)$
<code>shearY(a)</code>	inclina: ogni punto si sposta in verticale di $x \cdot \tan(a)$
<code>applyMatrix(...)</code>	applica una matrice affine data dall'utente
<code>pushMatrix()</code> / <code>popMatrix()</code>	salva e ripristina la matrice corrente
<code>resetMatrix()</code> / <code>printMatrix()</code>	ritorna all'identità / stampa la matrice nella console

Tabella 1. Le funzioni di trasformazione di Processing.

La rotazione è in senso orario perché l'asse y punta verso il basso. Gli angoli vanno dati in radianti: per i gradi si usa `radians(30)`.

`translate()`

Lo sketch seguente disegna lo stesso quadrato tre volte con le *stesse coordinate*; ciò che cambia è la traslazione intermedia. Le chiamate si **accumulano**: il terzo quadrato si trova a $(120 + 120, 0 + 20) = (240, 20)$ di distanza dal primo, non a $(120, 20)$.

```
size(400, 130);
background(255);
fill(0, 71, 255, 120);
stroke(0);

rect(20, 30, 60, 60);           // posizione originale
translate(120, 0);
rect(20, 30, 60, 60);           // spostato di 120 a destra
translate(120, 20);
rect(20, 30, 60, 60);           // spostamento accumulato: (240, 20)
```

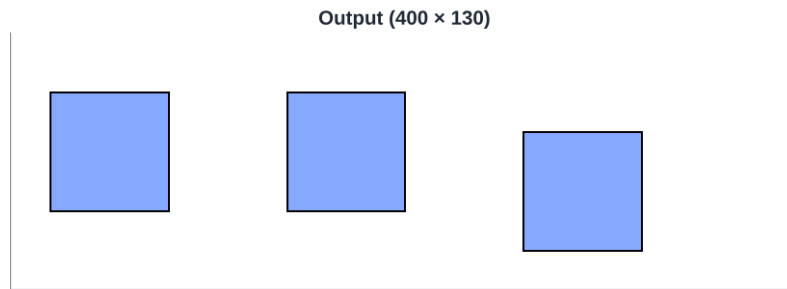


Figura 1. Tre quadrati disegnati con le stesse coordinate e traslazioni accumulate.

rotate(), scale(), shearX() e shearY()

Le altre trasformazioni agiscono in modo analogo, sempre rispetto all'**origine corrente**. Per questo, negli esempi, prima si trasla l'origine nell'angolo del quadrato e solo dopo si trasforma: così il quadrato si deforma attorno al proprio angolo in alto a sinistra. La funzione `resetMatrix()` riporta la matrice all'identità tra un riquadro e l'altro.

```
void base() {                                // riferimento grigio e poi quadrato colorato
  noFill();
  stroke(160);
  rect(0, 0, 70, 70);
  fill(0, 71, 255, 110);
  stroke(0);
}

void setup() {
  size(640, 170);
  background(255);

  translate(30, 30); base();
  scale(1.5, 0.8);                          // scala non uniforme
  rect(0, 0, 70, 70);

  resetMatrix(); translate(190, 30); base();
  shearX(PI / 6);                          // x' = x + y * tan(30 gradi)
  rect(0, 0, 70, 70);

  resetMatrix(); translate(350, 30); base();
  shearY(PI / 6);                          // y' = y + x * tan(30 gradi)
  rect(0, 0, 70, 70);

  resetMatrix(); translate(510, 30); base();
  rotate(PI / 6);                          // 30 gradi in senso orario
  rect(0, 0, 70, 70);
}
```

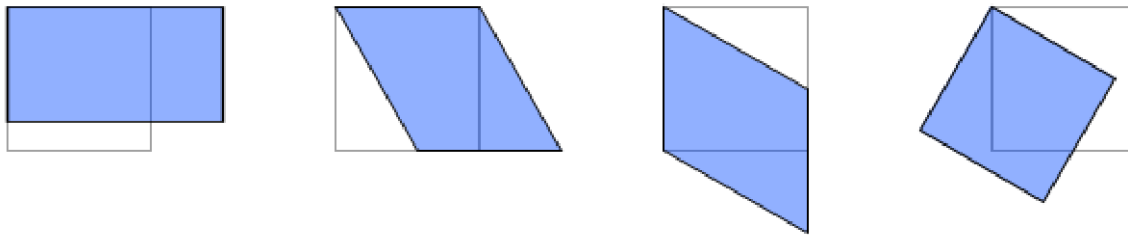


Figura 2. Da sinistra: scala non uniforme (1,5 × 0,8), shearX(30°), shearY(30°) e rotate(30°). Il contorno grigio è il quadrato originale.

ATTENZIONE

La scala moltiplica **anche lo spessore dei contorni**, perché è il sistema di coordinate a essere ingrandito. Per ruotare una figura attorno al suo **centro** si segue la ricetta già nota dalla teoria: si trasla l'origine nel centro, si ruota e si disegna la figura centrata nell'origine, per esempio con `translate(cx, cy); rotate(a); e rectMode(CENTER); rect(0, 0, w, h);`.

Le matrici e il legame con la teoria

Ogni funzione corrisponde a una matrice affine. Processing usa la convenzione dei **vettori colonna**: un punto (x, y), scritto in coordinate omogenee come colonna (con un 1 in terza posizione), viene moltiplicato a sinistra per la matrice. Della matrice 3 × 3 vengono mostrate solo le prime due righe, perché l'ultima è sempre (0, 0, 1).

translate			scale			rotate			shearX			shearY		
1	0	tx	sx	0	0	cos a	-sen a	0	1	tan a	0	1	0	0
0	1	ty	0	sy	0	sen a	cos a	0	0	1	0	tan a	1	0

Figura 3. Le prime due righe delle matrici di Processing (vettori colonna).

Nelle lezioni teoriche (capitolo 12 della dispensa di teoria) si usa invece la convenzione dei **vettori riga**, $[x \ y \ 1] = [v \ w \ 1] \cdot T$, con la traslazione nell'ultima riga della matrice.

Le due convenzioni descrivono le **stesse trasformazioni**: la matrice di Processing è la **trasposta** di quella della teoria (la rotazione, per esempio, ha $-\text{sen } a$ sopra la diagonale in una e sotto nell'altra). Cambia però il verso di lettura del prodotto: con i vettori riga $T = T_1 \cdot T_2$ applica prima T_1 e poi T_2 , con i vettori colonna $M = M_1 \cdot M_2$ applica prima M_2 . Nel codice questo significa che l'**ultima** trasformazione scritta è la **prima** ad agire sui punti.

L'ordine delle trasformazioni

Il prodotto di matrici non è commutativo, quindi cambiare l'ordine di `translate()` e `rotate()` cambia il risultato. Ogni chiamata moltiplica a **destra** la matrice corrente per quella nuova: dopo `translate()` e poi `rotate()` la matrice è $M = T \cdot R$.

```

void setup() {
  size(400, 220);
  background(255);
  fill(0, 71, 255, 110);
  stroke(0);

  // A: prima translate(), poi rotate()
  translate(30, 20);
  rect(0, 0, 80, 40);      // senza rotazione, per riferimento
  rotate(PI / 6);
  rect(0, 0, 80, 40);
  println("A");
  printMatrix();

  // B: prima rotate(), poi translate()
  resetMatrix();
  rotate(PI / 6);
  translate(200, 20);
  rect(0, 0, 80, 40);
  println("B");
  printMatrix();
}

```

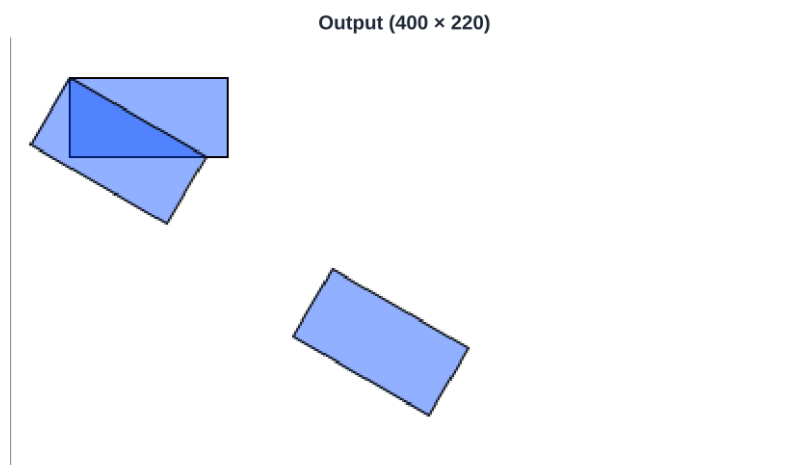


Figura 4. Caso A: il quadrato ruota attorno al punto (30, 20) (sopra, con il riferimento non ruotato). Caso B: prima ruota il sistema, poi lo spostamento di 200 avviene lungo l'asse ruotato.

La console mostra le due matrici:

```

A
0.8660 -0.5000 30.0000
0.5000 0.8660 20.0000
B
0.8660 -0.5000 163.2051
0.5000 0.8660 117.3205

```

Nel caso A la traslazione resta (30, 20). Nel caso B la matrice è $M = R \cdot T$ e la traslazione risultante è $R \cdot (200, 20)$, cioè $(200 \cos 30^\circ - 20 \sin 30^\circ, 200 \sin 30^\circ + 20 \cos 30^\circ) = (163,2; 117,3)$, come stampato: il quadrato finisce lontano dal punto dove sarebbe andato con l'ordine inverso.

pushMatrix() e popMatrix()

Quando si disegnano oggetti composti (una figura con parti che si muovono rispetto ad altre) servono trasformazioni «locali» che non disturbino il resto del disegno. `pushMatrix()` **salva** la matrice corrente in una pila e `popMatrix()` la **ripristina**; le due chiamate devono sempre essere in coppia. Le coppie si possono annidare, ottenendo una **gerarchia** di sistemi di coordinate.

L'esempio è un piccolo sistema solare: la terra gira attorno al sole, la luna attorno alla terra. Le istruzioni della luna sono scritte *dentro* il sistema della terra, quindi la luna segue la terra senza che si debba calcolare nulla. Il quadrato nero in basso, disegnato dopo i `popMatrix()`, non risente delle rotazioni.

```
void draw() {
  background(255);
  translate(width / 2, height / 2);      // origine al centro
  if (frameCount == 1) println("centro", screenX(0, 0), screenY(0, 0));
  fill(255, 190, 0);
  ellipse(0, 0, 50, 50);                  // sole

  pushMatrix();                          // salva la matrice corrente
  rotate(frameCount * 0.05);
  translate(110, 0);
  fill(0, 71, 255);
  ellipse(0, 0, 24, 24);                  // terra
  if (frameCount == 1) println("terra", screenX(0, 0), screenY(0, 0));

  pushMatrix();
  rotate(frameCount * 0.2);
  translate(28, 0);
  fill(120);
  ellipse(0, 0, 10, 10);                  // luna
  if (frameCount == 1) println("luna", screenX(0, 0), screenY(0, 0));
  popMatrix();                            // ripristina: sistema della terra

  popMatrix();                            // ripristina: sistema del sole
  if (frameCount == 1) println("dopo", screenX(0, 0), screenY(0, 0));
  fill(0);
  rect(-190, 90, 20, 20);                 // non risente delle rotazioni
}

void setup() {
  size(400, 240);
}
```

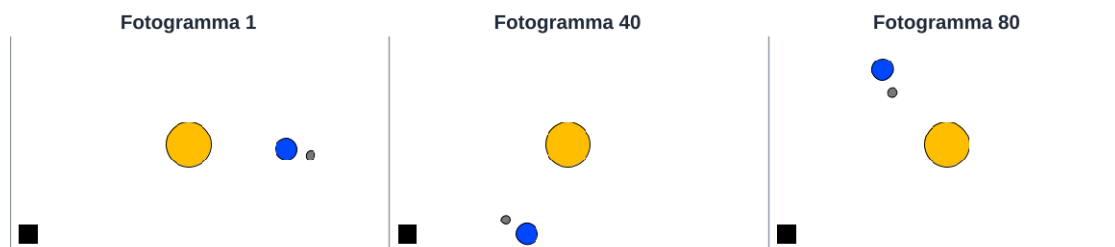


Figura 5. Tre fotogrammi del sistema solare (400 × 240): la terra orbita attorno al sole, la luna attorno alla terra, il quadrato nero resta fermo.

Nel primo fotogramma lo sketch stampa la posizione sullo schermo dell'origine del sistema corrente nei quattro momenti:

```
centro 200.0 120.0
terra 309.86252 125.49771
luna 336.99207 132.42502
dopo 200.0 120.0
```

Dopo `popMatrix()` l'origine è di nuovo al centro (200, 120). Con $a = 0,05$ radianti la terra sta in $(200 + 110 \cos a, 120 + 110 \sin a) = (309,86; 125,50)$, e la luna in $(200 + 110 \cos a + 28 \cos(a + 0,2), 120 + 110 \sin a + 28 \sin(a + 0,2)) = (336,99; 132,43)$: i valori coincidono con quelli calcolati a parte.

applyMatrix()

Con `applyMatrix(n00, n01, n02, n10, n11, n12)` si applica una matrice affine qualsiasi, indicando i sei coefficienti delle prime due righe, **per righe**. Il punto (x, y) viene portato in

$$x' = n_{00} \cdot x + n_{01} \cdot y + n_{02} \quad y' = n_{10} \cdot x + n_{11} \cdot y + n_{12}$$

Il collaudo è semplice: con la matrice (1, 2, 3; 4, 5, 6) il punto (1, 1) deve andare in $(1 + 2 + 3, 4 + 5 + 6) = (6, 15)$.

```
void setup() {
  size(100, 100);
  applyMatrix(1, 2, 3,
              4, 5, 6); // x' = 1x + 2y + 3, y' = 4x + 5y + 6
  println(screenX(1, 1), screenY(1, 1));
  printMatrix();
}
```

```
6.0 15.0
1.0000 2.0000 3.0000
4.0000 5.0000 6.0000
```

Il risultato conferma la convenzione a vettori colonna. Per usare in Processing una matrice della teoria (vettori riga) basta **trasporla**, scambiando righe e colonne, e prendere le prime due righe: la traslazione, che nella teoria sta nell'ultima riga, finisce nell'ultima colonna.

L'esempio seguente costruisce la rotazione di 30° con `rotate()` e con `applyMatrix()`: i due rettangoli (blu spesso e rosso sottile) coincidono perfettamente, e la console stampa due matrici identiche.

```

void setup() {
  size(200, 160);
  background(255);
  float a = radians(30);

  translate(60, 30);
  noFill();
  stroke(160);
  rect(0, 0, 80, 50);          // riferimento

  pushMatrix();
  stroke(0, 71, 255);
  strokeWeight(5);
  rotate(a);                    // rotazione con rotate()
  rect(0, 0, 80, 50);
  printMatrix();
  popMatrix();

  stroke(255, 0, 0);
  strokeWeight(1);
  applyMatrix(cos(a), -sin(a), 0,
              sin(a),  cos(a), 0); // la stessa con applyMatrix()
  rect(0, 0, 80, 50);
  printMatrix();
}

```

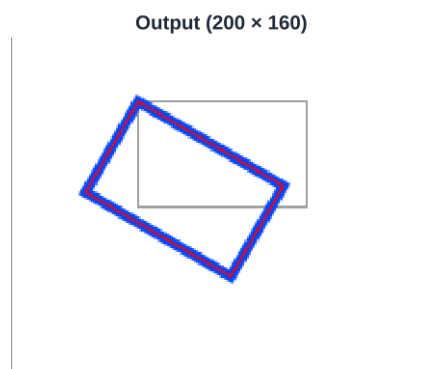


Figura 6. La stessa rotazione di 30° ottenuta con `rotate()` (blu spesso) e con `applyMatrix()` (rosso sottile): i contorni coincidono.

```

0.8660 -0.5000 60.0000
0.5000 0.8660 30.0000
0.8660 -0.5000 60.0000
0.5000 0.8660 30.0000

```

Con `applyMatrix()` si ottengono anche trasformazioni che le funzioni di base non offrono direttamente, come una **riflessione orizzontale**, $x' = -x + 80$, che si otterrebbe anche con `scale(-1, 1)` e una traslazione.

```

void setup() {
  size(200, 100);
  background(255);
  translate(60, 20);
  noFill();
  stroke(160);
  triangle(0, 0, 80, 0, 0, 50);          // riferimento
  stroke(0);
  fill(0, 71, 255, 110);
  applyMatrix(-1, 0, 80,                // x' = -x + 80
             0, 1, 0);                  // y' = y
  triangle(0, 0, 80, 0, 0, 50);
  println(screenX(0, 0), screenWidth(80, 0));
}

```

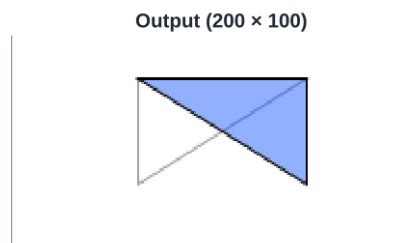


Figura 7. Riflessione orizzontale di un triangolo con `applyMatrix()`; in grigio l'originale.

La console stampa `140.0 60.0`: il lato sinistro del triangolo ($x = 0$) finisce nell'ascissa 140, cioè $60 + 80$, e il destro ($x = 80$) nell'ascissa 60, quella dell'origine traslata. Le due estremità si sono scambiate: la figura è specchiata.

DA RICORDARE

Le trasformazioni cambiano il sistema di coordinate, non gli oggetti già disegnati, e si **accumulano** fino alla fine del fotogramma o fino a `popMatrix()`. Gli angoli sono in radianti, la rotazione è oraria. L'ordine conta: l'ultima trasformazione scritta agisce per prima sui punti. `pushMatrix()/popMatrix()` isolano le trasformazioni locali. `applyMatrix()` usa vettori colonna, cioè la trasposta delle matrici a vettori riga del corso.

Classi, oggetti, ereditarietà e strutture dati

Definire nuovi tipi con le classi, creare oggetti, riusare il codice con l'ereditarietà e gestire collezioni di dati con array, matrici e ArrayList.

Classi e oggetti

Una **classe** descrive un nuovo tipo di dato: raggruppa i **campi** (le variabili che ne costituiscono lo stato) e i **metodi** (le funzioni che ne definiscono il comportamento). Un **oggetto** è una **istanza** della classe, creata con `new`: ogni oggetto ha la propria copia dei campi, mentre il codice dei metodi è condiviso. Le classi servono quando nello sketch compaiono più elementi con lo stesso comportamento (palline, particelle, personaggi), invece di tante variabili parallele come `x1`, `x2`, `x3`.

Elemento	Significato
Campo	variabile dichiarata dentro la classe (qui <code>x</code> , <code>y</code> , <code>raggio</code> , <code>vx</code> , <code>colore</code>)
Costruttore	metodo con lo stesso nome della classe e senza tipo di ritorno; viene eseguito da <code>new</code> e inizializza i campi
Metodo	funzione definita nella classe, che lavora sui campi dell'oggetto su cui è chiamata
Operatore punto	<code>a.x</code> legge un campo, <code>a.muovi()</code> chiama un metodo
<code>this</code>	riferimento all'oggetto corrente, utile quando un parametro ha lo stesso nome di un campo (<code>this.x = x</code>)

Tabella 1. Gli elementi di una classe.

Nell'esempio la classe `Pallina` rappresenta una pallina che rimbalza tra i bordi orizzontali della finestra: ne creiamo due con posizione, raggio, velocità e colore diversi. In `draw()` ognuna viene prima mossa e poi disegnata.

```

class Pallina {
    float x, y, raggio, vx;                // campi: lo stato dell'oggetto
    color colore;
    Pallina(float x0, float y0, float r0, float v0, color c0) { // costruttore
        x = x0; y = y0; raggio = r0; vx = v0; colore = c0;
    }

    void muovi() {                          // metodo: cambia lo stato
        x += vx;
        if (x < raggio || x > width - raggio) vx = -vx;
    }

    void mostra() {                         // metodo: disegna l'oggetto
        fill(colore);
        ellipse(x, y, 2 * raggio, 2 * raggio);
    }
}

Pallina a, b;                             // due oggetti, due istanze

void setup() {
    size(400, 160);
    a = new Pallina(40, 50, 20, 3, color(0, 71, 255));
    b = new Pallina(360, 110, 30, -2, color(255, 140, 0));
}

void draw() {
    background(255);
    a.muovi(); b.muovi();
    a.mostra(); b.mostra();
    if (frameCount % 40 == 0) println(frameCount, a.x, b.x);
}

```

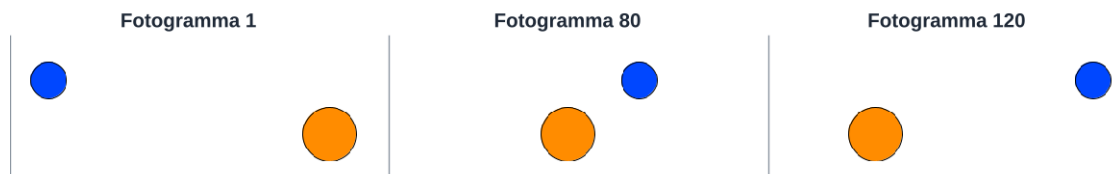


Figura 1. Le due palline (400 × 160 pixel) in tre momenti: ognuna ha posizione, raggio e velocità propri.

Ogni 40 fotogrammi lo sketch stampa la posizione orizzontale delle due palline:

```

40 160.0 280.0
80 280.0 200.0
120 364.0 120.0

```

I valori coincidono con una simulazione a parte (posizione 40 più 3 per fotogramma, con inversione quando supera 380): al fotogramma 80 la prima pallina arriva a 280; poco dopo il fotogramma 114 supera il bordo destro (380), la velocità cambia segno e al 120 la posizione è 364. La seconda, con velocità -2 e raggio 30, parte da 360 e rimbalza sul bordo sinistro prima del fotogramma 120, dove si trova a 120.

RIFERIMENTI

Una variabile di tipo classe non contiene l'oggetto ma un **riferimento** ad esso. L'assegnazione `Pallina c = a;` non crea una copia: `c` e `a` sono due nomi per lo stesso oggetto, e modificare l'uno modifica l'altro. Una variabile non ancora inizializzata vale `null`: usarne i campi o i metodi provoca un errore a runtime (`NullPointerException`). Per questo gli oggetti vanno creati con `new` in `setup()`, prima che `draw()` li usi.

Ereditarietà

L'**ereditarietà** permette di definire una classe a partire da un'altra: la **sottoclasse**, introdotta da `extends`, riceve tutti i campi e i metodi della **superclasse** e può aggiungerne di nuovi o **ridefinirli** (*override*). Nel costruttore della sottoclasse la prima istruzione, `super(...)`, richiama il costruttore della superclasse; con `super.metodo()` si può chiamare una versione ridefinita del metodo.

Nell'esempio `Forma` è la superclasse con la posizione, il colore e due metodi che le sottoclassi ridefiniscono. Le sottoclassi `Cerchio` e `Rettangolo` aggiungono i propri campi (il raggio, la larghezza e l'altezza) e disegnano la figura giusta.

```
class Forma {                                // superclasse
    float x, y;
    color colore;
    Forma(float x0, float y0, color c0) { x = x0; y = y0; colore = c0; }
    void mostra() { }                        // ridefinita dalle sottoclassi
    float area() { return 0; }
}

class Cerchio extends Forma {               // sottoclasse
    float r;
    Cerchio(float x0, float y0, float r0, color c0) {
        super(x0, y0, c0);                // costruttore della superclasse
        r = r0;
    }
    void mostra() { fill(colore); ellipse(x, y, 2 * r, 2 * r); }
    float area() { return PI * r * r; }
}

class Rettangolo extends Forma {
    float w, h;
    Rettangolo(float x0, float y0, float w0, float h0, color c0) {
        super(x0, y0, c0);
        w = w0; h = h0;
    }
    void mostra() { fill(colore); rect(x, y, w, h); }
    float area() { return w * h; }
}
```

Il vantaggio principale è il **polimorfismo**: un array (o una lista) di tipo `Forma` può contenere oggetti di qualsiasi sottoclasse, e chiamando `mostra()` su ciascuno viene eseguita la versione della sua classe, senza controlli del tipo.

```

Forma[] forme = new Forma[3]; // un array di Forma contiene sottoclassi

void setup() {
    size(400, 140);
    background(255);
    forme[0] = new Cerchio(60, 70, 40, color(0, 71, 255));
    forme[1] = new Rettangolo(140, 30, 120, 80, color(255, 140, 0));
    forme[2] = new Cerchio(330, 70, 25, color(0, 160, 90));
    for (Forma f : forme) {
        f.mostra(); // ogni oggetto usa il proprio mostra()
        println(f.area());
    }
}

```

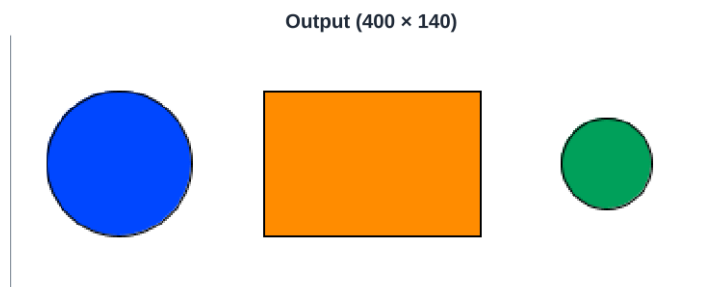


Figura 2. Due cerchi e un rettangolo, tutti disegnati con la chiamata `f.mostra()` su elementi di tipo `Forma`.

Nella console compaiono le tre aree, calcolate dal metodo `area()` di ciascuna sottoclasse: $\pi \cdot 40^2 = 5026,5483$ (con la precisione dei `float`), $120 \cdot 80 = 9600$ e $\pi \cdot 25^2 = 1963,4955$.

```

5026.5483
9600.0
1963.4955

```

Il vantaggio per chi legge il codice è che il ciclo `for (Forma f : forme)` non cambia se si aggiunge una nuova sottoclasse, per esempio `Triangolo`: basta scrivere la classe e inserirne un oggetto nell'array.

Strutture dati: array

Un **array** è una sequenza di elementi dello stesso tipo, con dimensione fissa e indici da 0 a `length - 1`. Si crea con `new tipo[n]` (gli elementi partono da 0, 0.0, false oppure null) o con un elenco di valori tra graffe. La proprietà `length` (senza parentesi) dà la dimensione. Superare l'indice massimo provoca l'errore `ArrayIndexOutOfBoundsException`.

Processing mette a disposizione funzioni per modificare gli array, che **restituiscono un nuovo array** e vanno quindi riassegnate: `append()`, `shorten()`, `expand()` (allunga l'array), `sort()`, `reverse()`, `concat()`, `subset()` (copia un'intera sequenza) e `split()` (da una stringa a un vettore di stringhe).

```

void setup() {
    int[] v = {5, 3, 8};                // array inizializzato
    println(v.length, v[0], v[v.length - 1]);
    v = append(v, 1);                  // nuovo array con un elemento in più
    v = sort(v);
    println(v);
    float[] w = new float[3];          // gli elementi partono da 0.0
    println(w[1]);
    String[] s = split("a,b,c", ',');
    println(s.length, s[2]);
    int somma = 0;
    for (int x : v) somma += x;         // for esteso
    println(somma);
}

```

```

3 5 8
[0] 1
[1] 3
[2] 5
[3] 8
0.0
3 c
17

```

Stampare un intero array con `println(v)` scrive un elemento per riga, preceduto dal proprio indice tra parentesi quadre. La somma finale vale $1 + 3 + 5 + 8 = 17$.

MATRICI (ARRAY BIDIMENSIONALI)

Un array di array è una **matrice**: `new int[6][10]` ha 6 righe e 10 colonne, e `g[i][j]` è l'elemento della riga `i` e colonna `j`. Dato che le righe sono array, `g.length` dà il numero di righe e `g[0].length` il numero di colonne. Se la matrice rappresenta una immagine, la riga è la coordinata `y` e la colonna è la coordinata `x`: l'ordine degli indici è `g[y][x]`, non `g[x][y]`. I due cicli annidati scorrono tutta la matrice, con il ciclo esterno sulle righe.

```

int[][] g = new int[6][10];           // 6 righe, 10 colonne

void setup() {
    size(300, 180);
    background(255);
    noStroke();
    for (int i = 0; i < g.length; i++) { // i = riga, y
        for (int j = 0; j < g[i].length; j++) { // j = colonna, x
            g[i][j] = (i * j) % 4;
            fill(255 - g[i][j] * 85);
            rect(j * 30, i * 30, 30, 30);
        }
    }
    println(g.length, g[0].length);
    println(g[2][3], g[5][9]);
}

```

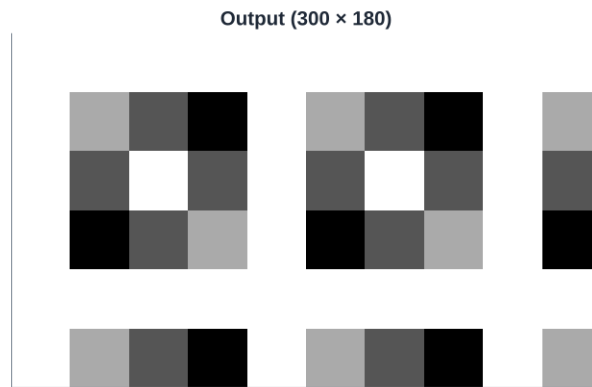


Figura 3. La matrice 6×10 con valori $(i \cdot j) \bmod 4$, disegnata come celle di 30 pixel con tonalità di grigio decrescente al crescere del valore (bianco = 0, nero = 3).

```
6 10
2 1
```

La prima riga contiene le dimensioni (6 righe, 10 colonne). La seconda riga stampa `g[2][3]` e `g[5][9]`, uguali a $(2 \cdot 3) \bmod 4 = 2$ e $(5 \cdot 9) \bmod 4 = 1$.

Strutture dati: ArrayList

Quando non si sa in anticipo quanti elementi serviranno si usa una **ArrayList**, una lista di oggetti che cresce e si accorcia. Si dichiara indicando il tipo degli elementi tra parentesi angolari. I metodi principali sono:

Metodo	Effetto
<code>add(x)</code>	aggiunge l'elemento <code>x</code> in coda
<code>get(i)</code>	restituisce l'elemento di indice <code>i</code>
<code>set(i, x)</code>	sostituisce l'elemento di indice <code>i</code>
<code>remove(i)</code>	elimina l'elemento di indice <code>i</code> (gli altri scalano)
<code>size()</code>	numero di elementi presenti
<code>clear()</code>	svuota la lista

Tabella 2. I metodi principali di ArrayList.

L'ArrayList contiene solo **oggetti** (per i numeri esistono `IntList` e `FloatList`, oppure le classi `Integer` e `Float`). Quando si elimina mentre si scorre la lista, conviene scorrerla **dalla fine**: eliminare l'elemento di indice `i` sposta quelli successivi, quindi con un ciclo in avanti si salterebbe un elemento.

L'esempio è una pioggia di gocce: un clic aggiunge una goccia nel punto del cursore; ogni goccia cade e, appena esce dalla finestra, viene tolta dalla lista.

```

class Goccia {
    float x, y;
    Goccia(float x0, float y0) { x = x0; y = y0; }
    void muovi() { y += 2; }
    boolean fuori() { return y > height + 8; }
    void mostra() { fill(0, 71, 255); ellipse(x, y, 16, 16); }
}

ArrayList<Goccia> lista = new ArrayList<Goccia>(); // lunghezza variabile

void setup() {
    size(300, 160);
}

void draw() {
    background(255);
    for (int i = lista.size() - 1; i >= 0; i--) { // dal fondo: rimozione sicura
        Goccia g = lista.get(i);
        g.muovi();
        g.mostra();
        if (g.fuori()) {
            lista.remove(i);
            println("rimossa, restano", lista.size());
        }
    }
}

void mousePressed() {
    lista.add(new Goccia(mouseX, mouseY));
    println("aggiunta, totale", lista.size());
}

```



Figura 4. Tre gocce aggiunte con tre clic simulati (300 × 160 pixel), mentre cadono.

La console conferma il comportamento: tre aggiunte (totale 1, 2, 3) e tre rimozioni, dalla goccia più in basso a quella più in alto:

```

aggiunta, totale 1
aggiunta, totale 2
aggiunta, totale 3
rimossa, restano 2
rimossa, restano 1
rimossa, restano 0

```

DA RICORDARE

Una classe raggruppa dati e comportamento; un oggetto si crea con `new` e si usa con l'operatore punto. Una sottoclasse (`extends`) eredita e può ridefinire; con `super` si richiama il costruttore della superclasse. Gli array hanno dimensione fissa e indici da 0; le matrici si indicizzano `[y][x]`; `ArrayList` cresce e si accorcia. Le variabili di tipo oggetto sono riferimenti.

Immagini e pixel: PImage

La classe PImage, loadImage(), createImage(), get(), set(), copy(), loadPixels(), updatePixels(), pixels, red(), green(), blue(), save() e saveFrame().

La classe PImage

Le immagini in Processing sono oggetti della classe **PImage**. Un oggetto PImage contiene la larghezza (width), l'altezza (height) e il vettore pixels[] con i colori di tutti i pixel. Per caricare un file si usa loadImage("nome.png"): il file deve trovarsi nella cartella data dello sketch, i formati supportati sono PNG, JPG, GIF e TGA. Se il file non viene trovato la funzione restituisce null e la prima operazione sull'immagine provoca un errore.

Con image(img, x, y) l'immagine si disegna nella finestra con l'angolo in alto a sinistra in (x, y); con image(img, x, y, w, h) la si ridimensiona a w × h. Il cambio di dimensione è fatto da Processing con una interpolazione, quindi non è un'operazione sui pixel dell'immagine originale: img non cambia.

```
PImage foto, astro;

void setup() {
  size(512, 256);
  foto = loadImage("camera.png");           // il file sta nella cartella data
  astro = loadImage("astro.png");
  println(foto.width, foto.height, astro.width, astro.height);
  image(foto, 0, 0, 256, 256);              // disegnata ridotta a 256 x 256
  image(astro, 256, 0, 256, 256);
}
```

Output (512 × 256)



Figura 1. Le due immagini del corso, entrambe di 512 × 512 pixel, disegnate ridotte a 256 × 256.

La console stampa 512 512 512 512: le due immagini hanno 512 × 512 pixel. L'immagine del fotografo è in toni di grigio (8 bit per pixel nel file), quella dell'astronauta è a colori; in Processing, dopo il caricamento, tutte le immagini diventano a 32 bit.

IL FORMATO INTERNO DEI PIXEL

Ogni pixel di un PImage è un intero color a 32 bit, in formato **ARGB** (capitolo 2): per un grigio le tre componenti sono uguali, per esempio 210 diventa FFD2D2D2, cioè -2960686. Per questo, nei capitoli successivi, per leggere il livello di grigio di un'immagine in toni di grigio basta `red(c)`.

get() e set(): leggere e scrivere un pixel

Chiamata	Effetto
<code>img.get(x, y)</code>	restituisce il colore del pixel (x, y)
<code>img.set(x, y, c)</code>	scrive il colore c nel pixel (x, y)
<code>img.get(x, y, w, h)</code>	restituisce un nuovo PImage con il ritaglio di w × h pixel con angolo in (x, y)
<code>img.get()</code>	restituisce una copia dell'intera immagine
<code>get(x, y) / set(x, y, c)</code>	stesse operazioni sulla finestra di disegno

Tabella 1. Le funzioni di lettura e scrittura dei pixel.

```
PImage img;

void setup() {
  size(100, 100);
  img = loadImage("camera.png");
  color c = img.get(100, 50);           // pixel con x = 100, y = 50
  println(red(c), green(c), blue(c));   // lo stesso colore come intero ARGB
  println(c, hex(c));

  img.set(100, 50, color(255, 0, 0));   // sovrascrive il pixel
  println(hex(img.get(100, 50)));

  println(img.get(600, 600));           // fuori dall'immagine: vale 0

  PImage taglio = img.get(160, 96, 64, 64); // ritaglio: x, y, larghezza, altezza
  println(taglio.width, taglio.height);
  println(hex(taglio.get(0, 0)), hex(img.get(160, 96)));
}
```

Il pixel (100, 50) della fotografia ha il valore 210 (lo si può verificare leggendo direttamente il file con un altro programma). Output della console:

```
210.0 210.0 210.0
-2960686 FFD2D2D2
FFFF0000
0
64 64
FFD2D2D2 FFD2D2D2
```

- la prima riga contiene i tre canali, tutti 210: `red()`, `green()` e `blue()` restituiscono un `float`, per questo compare 210.0 e non 210;
- la seconda mostra lo stesso colore come intero ARGB (-2960686) e in esadecimale (FFD2D2D2);
- dopo `set()` il pixel diventa rosso opaco (FFFF0000);

- una lettura fuori dall'immagine non dà errore ma restituisce 0 (nero trasparente);
- il ritaglio `img.get(160, 96, 64, 64)` ha 64×64 pixel, e il suo pixel (0, 0) coincide con il pixel (160, 96) dell'originale.

Le funzioni `get()` e `set()` sono comode ma **lente**, perché a ogni chiamata controllano i limiti e convertono il colore. Per elaborare tutta una immagine si usa invece direttamente il vettore `pixels`.

Il vettore `pixels[]`: `loadPixels()` e `updatePixels()`

Il vettore `pixels` contiene i pixel dell'immagine **riga dopo riga**, dalla prima riga (in alto) all'ultima, e in ciascuna riga da sinistra a destra: è un unico vettore monodimensionale di `width × height` elementi. Il pixel di coordinate (x, y) ha quindi indice

$$i = y \cdot \text{larghezza} + x \quad x = i \bmod \text{larghezza}, y = i / \text{larghezza} \text{ (divisione intera)}$$

Il pixel (0, 0) è il primo elemento, il pixel (`larghezza - 1`, 0) è l'ultimo della prima riga e il pixel (0, 1) è l'elemento di indice `larghezza`.

Il vettore non è sempre aggiornato: prima di leggerlo si chiama `img.loadPixels()`, che lo riempie con i pixel dell'immagine, e dopo averlo modificato si chiama `img.updatePixels()` per rendere effettive le modifiche. Per la finestra di disegno esistono le stesse funzioni senza prefisso, `loadPixels()` e `updatePixels()`, e un vettore globale `pixels`.

L'esempio crea una **nuova immagine vuota** con `createImage(w, h, RGB)` e vi scrive la fotografia riflessa orizzontalmente: il pixel (x, y) della sorgente va nel pixel (`w - 1 - x`, y) della destinazione.

```
PImage img, specchio;

void setup() {
  size(1024, 512);
  img = loadImage("camera.png");
  specchio = createImage(img.width, img.height, RGB); // immagine vuota
  img.loadPixels(); // riempie il vettore pixels
  specchio.loadPixels();
  for (int y = 0; y < img.height; y++) {
    for (int x = 0; x < img.width; x++) {
      int i = y * img.width + x; // indice lineare di (x, y)
      int j = y * img.width + (img.width - 1 - x); // pixel speculare
      specchio.pixels[j] = img.pixels[i];
    }
  }
  specchio.updatePixels(); // rende effettive le modifiche
  image(img, 0, 0);
  image(specchio, 512, 0);
  println(img.pixels.length);
}
```



Figura 2. A sinistra l'immagine originale, a destra la copia riflessa scritta pixel per pixel in una nuova immagine.

La console stampa 262144, cioè 512×512 elementi. Il risultato è stato confrontato con la stessa operazione calcolata a parte (inversione dell'ordine delle colonne): le due immagini coincidono in tutti i pixel.

ESTRARRE E COMPORRE I CANALI

Dato un colore c , le componenti si ottengono con `red(c)`, `green(c)` e `blue(c)` (restituiscono un float tra 0 e 255), oppure con operazioni sui bit, più rapide: $(c \gg 16) \& 0xFF$ per il rosso, $(c \gg 8) \& 0xFF$ per il verde e $c \& 0xFF$ per il blu (con $c \ggg 24$ la trasparenza). Per costruire un colore si usa `color(r, g, b)` (che accetta anche valori float e li limita tra 0 e 255), oppure, a mano, `0xFF000000 | (r << 16) | (g << 8) | b`.

Modo	Contenuto dei pixel	Uso
RGB	colori opachi (alpha sempre 255)	fotografie ed elaborazioni, il caso più comune
ARGB	colori con trasparenza	immagini con sfondo trasparente
ALPHA	solo la trasparenza (255 = opaco)	maschere

Tabella 2. Il terzo argomento di `createImage()`.

Per convertire un colore in un livello di grigio si usa la luminanza della teoria (capitolo 17), $0.299 * \text{red}(c) + 0.587 * \text{green}(c) + 0.114 * \text{blue}(c)$; per un'immagine che è già in toni di grigio basta uno qualsiasi dei tre canali.

copy(): copiare e ingrandire una zona

La funzione `copy(sx, sy, sw, sh, dx, dy, dw, dh)` copia un rettangolo di pixel in un altro rettangolo, e se i due hanno dimensioni diverse lo adatta (ingrandisce o riduce). Si può usare sulla finestra, con `copy(...)`, oppure tra immagini con `copy(src, sx, sy, sw, sh, dx, dy, dw, dh)` o `img.copy(...)`.

```

PImage img;

void setup() {
  size(512, 256);
  img = loadImage("camera.png");
  background(255);
  image(img, 0, 0, 256, 256);
  noFill();
  stroke(255, 0, 0);
  rect(80, 48, 32, 32);          // zona (160, 96, 64, 64) a meta'
  copy(img, 160, 96, 64, 64, 256, 0, 256, 256);    // sorgente, destinazione
}

```

Output (512 × 256)



Figura 3. A sinistra l'immagine ridotta con il riquadro rosso (zona di 64×64 pixel dell'originale), a destra la stessa zona ingrandita 4 volte con `copy()`.

Nell'ingrandimento $64 \rightarrow 256$ pixel i valori intermedi non sono ripetuti a blocchi ma calcolati per interpolazione: l'orecchio risulta sfumato, non «a scalini». Per applicare un metodo di interpolazione particolare (replication, bilineare, bicubica) lo si scrive direttamente, come si farà nel capitolo sulla interpolazione.

Salvare le immagini: `save()` e `saveFrame()`

Funzione	Cosa salva
<code>save("file.png")</code>	l'intera finestra di disegno
<code>saveFrame("nome-####.png")</code>	la finestra, con il numero del fotogramma al posto dei cancelletti
<code>img.save("file.png")</code>	un oggetto <code>PImage</code>
<code>saveFrame()</code>	senza argomenti: nome predefinito <code>screen-####.tif</code>

Tabella 3. Le funzioni di salvataggio.

Il formato è scelto dall'estensione (`.png`, `.jpg`, `.tif` o `.tga`). I file vengono scritti nella **cartella dello sketch**, non in data: quest'ultima si usa solo per i file da caricare. Nell'esempio, al terzo fotogramma si salvano tre file.

```

PImage img;

void setup() {
  size(200, 200);
  img = loadImage("camera.png");
}

void draw() {
  image(img, 0, 0, width, height);
  noStroke();
  fill(255, 0, 0);
  ellipse(frameCount * 20, 100, 30, 30);
  if (frameCount == 3) {
    save("fotogramma.png");           // salva l'intera finestra
    saveFrame("fr-####.png");        // numera con il fotogramma corrente
    img.save("originale.png");         // salva un oggetto PImage
  }
}

```

Dopo l'esecuzione la cartella dello sketch contiene i file `fotogramma.png`, `fr-0003.png`, `originale.png`. Il numero 0003 è il fotogramma al momento della chiamata: i cancelletti vengono sostituiti dal valore di `frameCount` con tanti zeri iniziali quanti sono i cancelletti.

Lo schema degli algoritmi sulle immagini

Quasi tutti gli algoritmi dei capitoli seguenti hanno la stessa forma: si carica l'immagine, si crea una immagine di uscita delle stesse dimensioni (o di dimensioni calcolate), si aprono i due vettori di pixel, si scorrono le righe e le colonne leggendo dall'ingresso e scrivendo nell'uscita, e infine si aggiorna e si mostra l'uscita.

```

PImage in = loadImage("camera.png");
PImage out = createImage(in.width, in.height, RGB);
in.loadPixels();
out.loadPixels();
for (int y = 0; y < in.height; y++) {
  for (int x = 0; x < in.width; x++) {
    int i = y * in.width + x;
    float v = red(in.pixels[i]);           // livello di grigio
    // ... calcolo del nuovo valore w ...
    out.pixels[i] = color(w);
  }
}
out.updatePixels();
image(out, 0, 0);

```

Scrivere sempre in una immagine **distinta** dall'ingresso è necessario per gli operatori che usano i pixel vicini (filtri, convoluzione): se si sovrascrivesse l'ingresso, i pixel successivi verrebbero calcolati a partire da valori già modificati.

DA RICORDARE

Un `PImage` ha `width`, `height` e `pixels[]`; i pixel sono interi ARGB e la riga `y` è lunga `width`. Il pixel (x, y) è l'elemento $y * width + x$ del vettore. Si chiama `loadPixels()` prima di leggere o scrivere pixel e `updatePixels()` dopo averlo modificato. `get()` e `set()` sono semplici ma lenti. I file si caricano da data e si salvano nella cartella dello sketch.

Quantizzazione, interpolazione e PSNR

Il calcolo del PSNR, la quantizzazione uniforme e logaritmica dei livelli di grigio e l'ingrandimento di una immagine per replication.

Misurare la qualità: il PSNR

Per giudicare in modo oggettivo quanto un'immagine elaborata somiglia all'originale si usa il **PSNR** (capitolo 13 della dispensa di teoria). Con a l'immagine originale, b quella elaborata, entrambe di n pixel, e il massimo valore 255:

$$\text{MSE} = (1/n) \cdot \sum (a_i - b_i)^2 \quad \text{PSNR} = 10 \cdot \log_{10}(255^2 / \text{MSE}) \text{ dB}$$

La funzione seguente lavora su immagini in toni di grigio e la useremo in tutti i capitoli seguenti. L'accumulo dell'errore è fatto con una variabile `double` per non perdere precisione sommando 262 144 termini; se le due immagini sono identiche l'MSE è nullo e il risultato è infinito.

```
float psnr(PImage a, PImage b) {
  a.loadPixels();
  b.loadPixels();
  double mse = 0;
  for (int i = 0; i < a.pixels.length; i++) {
    double d = red(a.pixels[i]) - red(b.pixels[i]); // differenza dei livelli di grigio
    mse += d * d;
  }
  mse /= a.pixels.length; // errore quadratico medio
  if (mse == 0) return Float.POSITIVE_INFINITY; // immagini identiche
  return (float) (10 * Math.log10(255.0 * 255.0 / mse));
}

void setup() {
  size(100, 100);
  PImage a = loadImage("camera.png");
  PImage b = a.get(); // copia
  println(psnr(a, b));
  b.loadPixels();
  for (int i = 0; i < b.pixels.length; i++)
    b.pixels[i] = color(min(255, red(b.pixels[i]) + 10)); // 'luminosità' + 10
  b.updatePixels();
  println(psnr(a, b));
}
```

L'esempio confronta la fotografia con una copia identica e con una copia più luminosa di 10 livelli (con limite a 255). Sulla console compaiono `Infinity` e `28.146307`: il valore infinito per l'immagine identica e 28,146 dB per la copia più luminosa. Il secondo numero è stato controllato con un calcolo indipendente sugli stessi dati, ed è uguale fino alla sesta cifra.

Quantizzazione uniforme

La **quantizzazione uniforme** riduce i livelli di grigio da $N = 256$ a K dividendo l'intervallo $0 \dots 255$ in K parti uguali (capitolo 9 della teoria). Il livello di uscita è

$$L' = \lfloor L \cdot K / N \rfloor$$

In Java la divisione tra interi scarta già i decimali, quindi $v * K / 256$ è esattamente la formula. I livelli $0 \dots K - 1$ sono poco adatti alla visualizzazione, perché il nero e un grigio quasi nero sembrerebbero uguali: per mostrare il risultato ogni livello viene riportato al **centro del proprio intervallo**, $(L' + 0,5) \cdot N / K$, che è anche il valore che minimizza l'errore.

```
PImage quantizza(PImage in, int K) {
    PImage out = createImage(in.width, in.height, RGB);
    in.loadPixels();
    out.loadPixels();
    for (int i = 0; i < in.pixels.length; i++) {
        int v = (int) red(in.pixels[i]); // livello 0 ... 255
        int q = v * K / 256;           // livello quantizzato 0 ... K - 1
        int g = (int) ((q + 0.5) * 256 / K); // centro dell'intervallo (per mostrarlo)
        out.pixels[i] = color(g);
    }
    out.updatePixels();
    return out;
}
```

La funzione si usa su diversi numeri di livelli e per ciascuno si calcola il PSNR rispetto all'originale:

```
void setup() {
    size(100, 100);
    PImage in = loadImage("camera.png");
    int[] livelli = {16, 8, 4, 2};
    for (int K : livelli) {
        PImage out = quantizza(in, K);
        out.save("q" + K + ".png");
        println(K, psnr(in, out)); // psnr() come nel listato precedente
    }
}
```

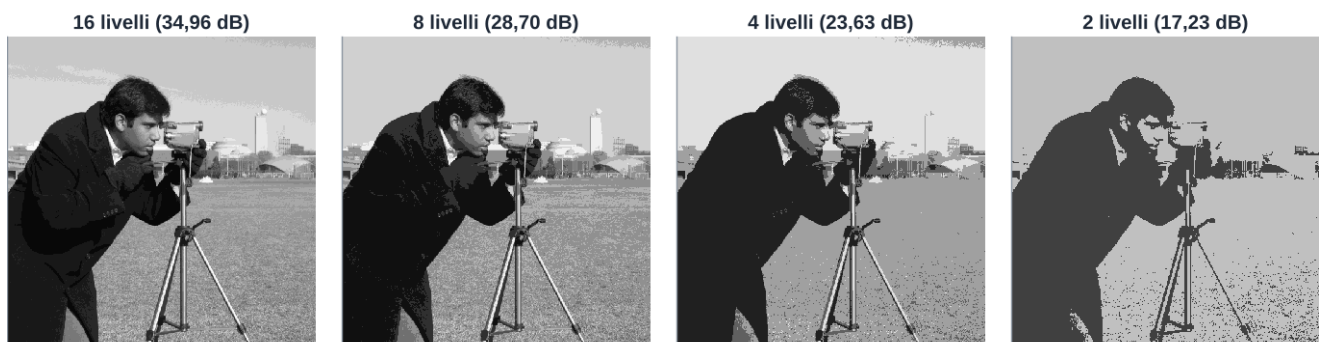


Figura 1. La fotografia quantizzata con 16, 8, 4 e 2 livelli (4, 3, 2 e 1 bit): compaiono falsi contorni nelle zone che prima erano sfumate.

```
16 34.95681
8 28.70063
4 23.627722
2 17.23452
```

Il PSNR passa da 34,96 dB con 16 livelli a 28,70 con 8, 23,63 con 4 e 17,23 con 2. Ogni bit in meno costa da 5 a 6 dB circa, in accordo con la regola della teoria (≈ 6 dB per bit, valida per una distribuzione dei livelli piuttosto uniforme; qui i livelli non lo sono). Le quattro immagini prodotte dallo sketch sono state confrontate pixel per pixel con quelle calcolate separatamente, e sono identiche.

Quantizzazione logaritmica

Nella quantizzazione **logaritmica** i livelli non sono equidistanti: gli intervalli sono stretti per i toni scuri e larghi per quelli chiari, come nella sensibilità dell'occhio. La formula del corso è

$$L' = \lfloor \log_2(L) \cdot K / \log_2(N) \rfloor \quad (\text{con } L' = 0 \text{ per } L = 0)$$

Per $K = 8$ e $N = 256$ diventa $L' = \lfloor \log_2(L) \rfloor$: i livelli sono gli intervalli 0 ... 1, 2 ... 3, 4 ... 7, 8 ... 15, ..., 128 ... 255. In Processing la funzione `log()` calcola il logaritmo in base 2 come $\log(v) / \log(2)$, perché esiste solo il logaritmo naturale (`log`). La funzione `centroLog()` riporta ogni livello al centro del proprio intervallo di ingresso.

```
int livelloLog(int v, int K) {
    if (v == 0) return 0;                // log(0) non e' definito
    return (int) (log(v) / log(2) * K / 8); // floor(log2(L) * K / log2(256))
}

int centroLog(int q) {                  // centro dell'intervallo (K = 8)
    if (q == 0) return 0;
    int basso = (int) pow(2, q);
    int alto = (int) pow(2, q + 1) - 1;
    return (basso + alto) / 2;
}
```

Per confrontare i due metodi usiamo una immagine **scura**, ottenuta elevando al cubo i livelli normalizzati ($v = 255 \cdot t^3$ con $t = L / 255$): è il caso in cui la quantizzazione logaritmica dovrebbe dare il meglio. Lo sketch stampa prima i livelli di alcuni valori campione, poi salva le tre immagini e i due PSNR rispetto alla immagine scura.

```
void setup() {
    size(100, 100);
    int[] campioni = {0, 1, 2, 3, 4, 7, 8, 15, 16, 127, 128, 255};
    String riga = "";
    for (int v : campioni) riga += livelloLog(v, 8) + " ";
    println(riga);

    PImage in = loadImage("camera.png");
    PImage buio = createImage(in.width, in.height, RGB);
    PImage uni = createImage(in.width, in.height, RGB);
    PImage log = createImage(in.width, in.height, RGB);
    in.loadPixels(); buio.loadPixels(); uni.loadPixels(); log.loadPixels();
    for (int i = 0; i < in.pixels.length; i++) {
        float t = red(in.pixels[i]) / 255.0;
        int v = (int) (255 * t * t * t);          // immagine scura: v = 255 * t^3
        buio.pixels[i] = color(v);
        uni.pixels[i] = color((v * 8 / 256 * 32) + 16); // 8 livelli uniformi
        log.pixels[i] = color(centroLog(livelloLog(v, 8))); // 8 livelli logaritmici
    }
    buio.updatePixels(); uni.updatePixels(); log.updatePixels();
    buio.save("buio.png"); uni.save("uni.png"); log.save("log.png");
    println(psnr(buio, uni), psnr(buio, log));
}
```



Figura 2. A sinistra l'immagine scura (256 livelli), al centro la quantizzazione uniforme e a destra quella logaritmica, entrambe a 8 livelli.

```
0 0 1 1 2 2 3 3 4 6 7 7
26.9068 20.450726
```

La prima riga della console riporta i livelli dei valori campione 0, 1, 2, 3, 4, 7, 8, 15, 16, 127, 128, 255: 0 0 1 1 2 2 3 3 4 6 7 7. Coincidono con gli intervalli della teoria (0 e 1 nel livello 0, 2 e 3 nel livello 1, 4 e 7 nel livello 2 e così via) e con quelli calcolati a parte per **tutti** i 256 valori e per 2, 4, 8 e 16 livelli.

La seconda riga contiene i PSNR: 26,91 dB per la quantizzazione uniforme e 20,45 dB per quella logaritmica. Il risultato sorprende solo a prima vista. Il PSNR misura l'errore **assoluto** pixel per pixel, e la quantizzazione logaritmica, pur conservando meglio la struttura nelle zone scure (nella figura il cappotto e il prato mostrano più dettagli), commette errori assoluti più grandi nelle zone chiare, dove gli intervalli sono larghissimi (da 128 a 255). Il PSNR è un indice comodo ma non rappresenta la qualità percepita.

Interpolazione: replication

Per ingrandire una immagine bisogna inventare i pixel mancanti. Con la **replication** (o *nearest neighbour*, vicino più prossimo) ogni nuovo pixel prende il valore del pixel originale più vicino: per uno zoom di 2 ogni pixel viene ripetuto in un blocco 2×2 (capitolo 12 della teoria). Il pixel (x, y) della immagine ingrandita è quindi il pixel $(\lfloor x / 2 \rfloor, \lfloor y / 2 \rfloor)$ di quella piccola, ed è sufficiente la divisione intera.

Per valutare il metodo in modo oggettivo si **riduce** la fotografia a 256×256 con la media su blocchi 2×2 (con arrotondamento, $(\text{somma} + 2) / 4$), si ingrandisce di nuovo a 512×512 con la replication e si calcola il PSNR rispetto all'originale. Lo sketch comprende anche la funzione `psnr()`.

```

PImage riduci(PImage in) {                                // media su blocchi 2 x 2
    int w = in.width / 2, h = in.height / 2;
    PImage out = createImage(w, h, RGB);
    in.loadPixels();
    out.loadPixels();
    for (int y = 0; y < h; y++) {
        for (int x = 0; x < w; x++) {
            int somma = 0;
            for (int dy = 0; dy < 2; dy++)
                for (int dx = 0; dx < 2; dx++)
                    somma += (int) red(in.pixels[(2 * y + dy) * in.width + 2 * x + dx]);
            out.pixels[y * w + x] = color((somma + 2) / 4);    // media arrotondata
        }
    }
    out.updatePixels();
    return out;
}

PImage ingrandisci(PImage in) {                            // replication: ogni pixel diventa 2 x 2
    PImage out = createImage(2 * in.width, 2 * in.height, RGB);
    in.loadPixels();
    out.loadPixels();
    for (int y = 0; y < out.height; y++)
        for (int x = 0; x < out.width; x++)
            out.pixels[y * out.width + x] = in.pixels[(y / 2) * in.width + (x / 2)];
    out.updatePixels();
    return out;
}

```

```

void setup() {
    size(100, 100);
    PImage a = loadImage("camera.png");
    PImage piccola = riduci(a);                        // 512 x 512 -> 256 x 256
    PImage grande = ingrandisci(piccola);              // 256 x 256 -> 512 x 512
    println(piccola.width, grande.width);
    println(psnr(a, grande));
    a.get(196, 72, 80, 80).save("c_orig.png");
    grande.get(196, 72, 80, 80).save("c_rep.png");
}

```



Figura 3. Ritaglio di 80 × 80 pixel (capelli e fronte): a destra la fotografia ridotta e riportata a 512 × 512 con la replication, che produce i blocchi 2 × 2 ben visibili.

256 512
28.681484

Sulla console compaiono le dimensioni (256 e 512) e il PSNR di 28,68 dB. Lo stesso valore, 28,68 dB, si ottiene con il calcolo di riferimento usato nella dispensa di teoria per la replication (capitolo 12), e l'immagine ingrandita è identica pixel per pixel a quella calcolata con il metodo di riferimento.

Gli altri metodi del capitolo di teoria, bilineare e bicubica, si implementano nello stesso modo: per ogni pixel della immagine di uscita si calcola la posizione corrispondente nell'immagine di ingresso e si usano i vicini con i pesi del metodo. La replication è il più semplice perché usa un solo vicino e nessun peso.

DA RICORDARE

Il PSNR è $10 * \log_{10}(255 * 255 / \text{MSE})$, calcolato su tutti i pixel. La quantizzazione uniforme è $v * K / 256$ (divisione intera), quella logaritmica è $\log(v) / \log(2) * K / 8$. Per mostrare i livelli quantizzati li si riporta al centro dell'intervallo. La replication ripete i pixel: l'ingrandimento di un fattore 2 usa `in.pixels[(y / 2) * w + (x / 2)]`.

Operazioni puntuali, stretching ed equalizzazione

Le trasformazioni che dipendono solo dal valore del pixel: negativo, logaritmo e gamma con una LUT, istogramma, espansione del contrasto ed equalizzazione.

Operazioni puntuali e LUT

Un'operazione puntuale calcola il nuovo valore di un pixel soltanto a partire dal suo valore attuale, $g = T(f)$, senza guardare i pixel vicini né la posizione (capitolo 20 della dispensa di teoria). Dato che i livelli di grigio sono solo 256, conviene calcolare T una volta per tutti i valori e memorizzarla in una **tabella di corrispondenza** (*look up table*, LUT): un vettore di 256 interi in cui $\text{lut}[v]$ è il nuovo valore per il livello v . Applicarla all'immagine è poi un semplice ciclo con una lettura di vettore per pixel, comunque sia fatta la funzione.

Il codice seguente contiene la funzione `applica()`, che applica una LUT qualsiasi, e tre funzioni che costruiscono le LUT delle trasformazioni principali:

Trasformazione	Formula	Effetto
Negativo	$g = 255 - f$	inverte chiari e scuri
Logaritmo	$g = c \cdot \ln(1 + f)$, $c = 255 / \ln 256 \approx 45,98$	espande i toni scuri, comprime i chiari
Gamma	$g = 255 \cdot (f / 255)^{\gamma}$	γ minore di 1 schiarisce, γ maggiore di 1 scurisce

Tabella 1. Le tre trasformazioni puntuali del laboratorio.

```

PImage applica(PImage in, int[] lut) { // vale per qualsiasi trasformazione
    PImage out = createImage(in.width, in.height, RGB);
    in.loadPixels();
    out.loadPixels();
    for (int i = 0; i < in.pixels.length; i++)
        out.pixels[i] = color(lut[(int) red(in.pixels[i])]);
    out.updatePixels();
    return out;
}

int[] lutNegativo() {
    int[] t = new int[256];
    for (int v = 0; v < 256; v++) t[v] = 255 - v;
    return t;
}

int[] lutLog() {
    int[] t = new int[256];
    float c = 255 / log(256); // c = 255 / ln(256) = 45,98
    for (int v = 0; v < 256; v++) t[v] = constrain(round(c * log(1 + v)), 0, 255);
    return t;
}

int[] lutGamma(float g) {
    int[] t = new int[256];
    for (int v = 0; v < 256; v++) t[v] = constrain(round(255 * pow(v / 255.0, g)), 0, 255);
    return t;
}

```

```

void setup() {
    size(100, 100);
    PImage in = loadImage("camera.png");
    applica(in, lutNegativo()).save("neg.png");
    applica(in, lutLog()).save("log.png");
    applica(in, lutGamma(0.4)).save("g04.png");
    applica(in, lutGamma(2.5)).save("g25.png");
    int[] l = lutLog();
    println(l[0], l[7], l[63], l[255]);
    println(lutGamma(0.4)[64], lutGamma(2.5)[64]);
}

```



Figura 1. Le quattro trasformazioni applicate alla fotografia del corso: il negativo, il logaritmo e due correzioni gamma.

```
0 96 191 255
147 8
```

La prima riga della console riporta i valori della LUT del logaritmo per i livelli 0, 7, 63 e 255: 0 96 191 255. Sono i valori calcolati come esempio nel capitolo di teoria (96 per $f = 7$, 191 per $f = 63$). La seconda riga contiene la LUT gamma per il livello 64: 147 con $\gamma = 0,4$ e 8 con $\gamma = 2,5$. Tutte e quattro le immagini sono state confrontate con il calcolo di riferimento e coincidono in ogni pixel.

IMMAGINI A COLORI

Le operazioni puntuali si estendono alle immagini RGB applicando la stessa LUT ai tre canali separatamente (con il rischio di alterare le tinte) oppure alla sola luminosità. Nel codice basta leggere `red(c)`, `green(c)` e `blue(c)`, applicare `lut[ ]` a ciascuno e ricomporre con `color(r, g, b)`.

L'istogramma di una immagine

L'**istogramma** conta, per ogni livello di grigio, quanti pixel dell'immagine hanno quel livello (capitolo 19 della teoria). Il calcolo è un solo ciclo con un vettore di 256 contatori. La funzione `disegna()` costruisce un piccolo grafico da salvare o mostrare: l'altezza di ciascuna barra è proporzionale alla **radice quadrata** del contatore, scelta che rende visibili anche i livelli poco frequenti quando un valore (per esempio il nero) è molto più popolato degli altri.

```
int[] istogramma(PImage in) {
    int[] h = new int[256];
    in.loadPixels();
    for (int i = 0; i < in.pixels.length; i++)
        h[(int) red(in.pixels[i])]++;           // un pixel in piu' nel livello v
    return h;
}

PImage disegna(int[] h) {                       // grafico 256 x 100
    PGraphics g = createGraphics(256, 100);
    g.beginDraw();
    g.background(255);
    g.stroke(0, 71, 255);
    int massimo = max(h);
    for (int v = 0; v < 256; v++)
        g.line(v, 99, v, 99 - sqrt(h[v]) * 99.0 / sqrt(massimo)); // scala radice
    g.endDraw();
    return g;
}
```

La funzione `createGraphics()` crea un oggetto `PGraphics` separato dalla finestra, dove si può disegnare con gli stessi comandi (preceduti dal nome dell'oggetto) tra `beginDraw()` e `endDraw()`: `PGraphics` è anche un `PImage`, quindi si può restituire, mostrare con `image()` e salvare con `save()`.

Espansione del contrasto (stretching)

Quando l'istogramma occupa solo una parte della scala, l'immagine ha poco contrasto. L'espansione del contrasto trasforma in modo lineare il minimo osservato in 0 e il massimo in 255:

$$v_{\text{nuovo}} = 255 \cdot (v - v_{\text{min}}) / (v_{\text{max}} - v_{\text{min}})$$

La funzione `stiramento()` trova minimo e massimo con un primo ciclo e applica la formula con un secondo.

```

PImage stiramento(PImage in) {
    in.loadPixels();
    int minimo = 255, massimo = 0;
    for (int i = 0; i < in.pixels.length; i++) {
        int v = (int) red(in.pixels[i]);
        minimo = min(minimo, v);
        massimo = max(massimo, v);
    }
    PImage out = createImage(in.width, in.height, RGB);
    out.loadPixels();
    for (int i = 0; i < in.pixels.length; i++) {
        float v = red(in.pixels[i]);
        out.pixels[i] = color(round(255 * (v - minimo) / (massimo - minimo)));
    }
    out.updatePixels();
    return out;
}

```

Equalizzazione

L'**equalizzazione** rende l'istogramma il più possibile uniforme. Il nuovo valore del livello k è proporzionale alla frazione di pixel che hanno livello minore o uguale a k , cioè alla funzione di distribuzione cumulata dell'istogramma, riportata sulla scala 0 ... 255:

$$s_k = \text{arrotondamento di } 255 \cdot \left(\sum_{j=0 \dots k} h(j) \right) / (M \cdot N)$$

Nel codice la somma cumulata si costruisce con un solo ciclo sui livelli, e il risultato è una LUT da applicare con lo stesso ciclo dello stiramento. La somma è dichiarata long perché, per immagini molto grandi, supererebbe il massimo degli int.

```

PImage equalizza(PImage in) {
    int[] h = istogramma(in);
    int[] lut = new int[256];
    long somma = 0; // istogramma cumulato
    for (int k = 0; k < 256; k++) {
        somma += h[k];
        lut[k] = round((float) (255.0 * somma / in.pixels.length));
    }
    PImage out = createImage(in.width, in.height, RGB);
    out.loadPixels();
    for (int i = 0; i < in.pixels.length; i++)
        out.pixels[i] = color(lut[(int) red(in.pixels[i])]);
    out.updatePixels();
    return out;
}

```

Una prova completa

Per provare le due funzioni si fabbricano a partire dalla fotografia una immagine **poco contrastata**, con i livelli ristretti tra 85 e 170 ($v = 85 + 85 \cdot f / 255$), e una immagine **scura**, con $v = 255 \cdot (f / 255)^3$. Si stira la prima e si equalizza la seconda, e si salvano le immagini e i relativi istogrammi.

```

void setup() {
  size(100, 100);
  PImage in = loadImage("camera.png");
  PImage poco = createImage(in.width, in.height, RGB); // immagine poco contrastata
  PImage buio = createImage(in.width, in.height, RGB); // immagine scura
  in.loadPixels(); poco.loadPixels(); buio.loadPixels();
  for (int i = 0; i < in.pixels.length; i++) {
    float v = red(in.pixels[i]);
    poco.pixels[i] = color(round(85 + v * 85 / 255)); // valori da 85 a 170
    buio.pixels[i] = color((int) (255 * pow(v / 255, 3))); // v = 255 * t^3
  }
  poco.updatePixels(); buio.updatePixels();
  poco.save("poco.png");
  buio.save("buio.png");
  PImage s = stiramento(poco);
  PImage e = equalizza(buio);
  s.save("stirata.png");
  e.save("equalizzata.png");
  disegna(istogramma(poco)).save("h_poco.png");
  disegna(istogramma(s)).save("h_stirata.png");
  disegna(istogramma(buio)).save("h_buio.png");
  disegna(istogramma(e)).save("h_equalizzata.png");
  int[] h = istogramma(s);
  int usati = 0;
  for (int v = 0; v < 256; v++) if (h[v] > 0) usati++;
  println("livelli usati dopo lo stiramento:", usati);
}

```



Figura 2. A sinistra l'immagine con i livelli ristretti tra 85 e 170, a destra la stessa dopo l'espansione del contrasto.

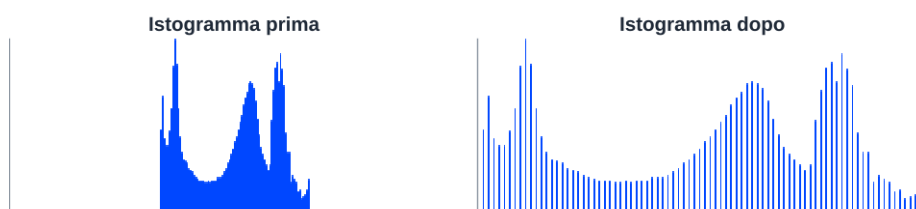


Figura 3. Gli istogrammi (altezze in scala radice): l'espansione sposta e allarga la distribuzione senza cambiarne la forma.

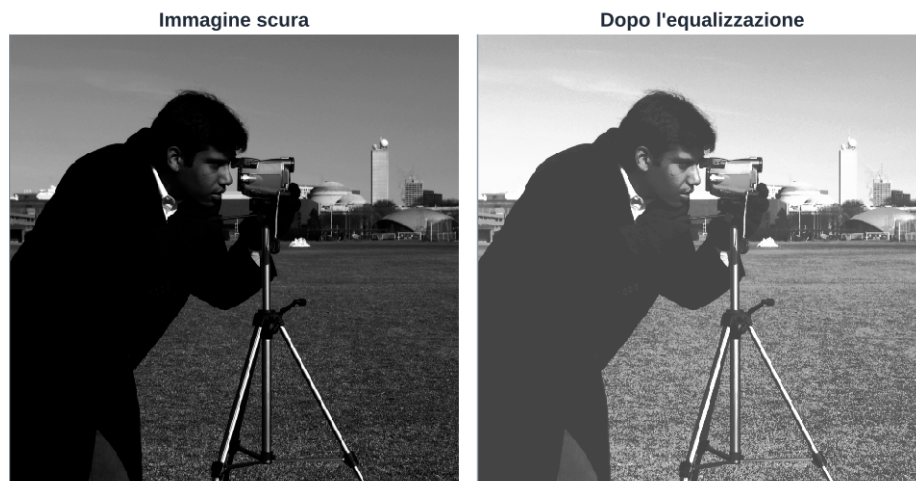


Figura 4. A sinistra l'immagine scura, a destra l'immagine equalizzata: nelle ombre e nel prato compaiono dettagli prima invisibili.

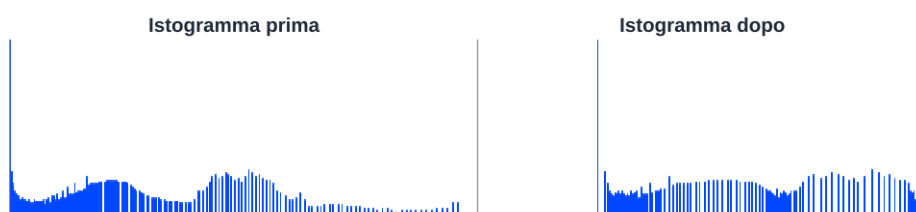


Figura 5. Gli istogrammi prima e dopo l'equalizzazione (scala radice): quasi tutti i livelli sono ora occupati, in modo più regolare.

livelli usati dopo lo stiramento: 86

La console conferma l'osservazione già fatta nel capitolo di teoria: dopo lo stiramento l'immagine usa ancora soltanto 86 livelli distinti (tutti i valori da 85 a 170 hanno un corrispondente), quindi l'istogramma ha un aspetto «a pettine» con barre distanziate. Lo stretching non crea informazione: ridistribuisce i livelli esistenti. L'immagine stirata e quella equalizzata sono identiche, pixel per pixel, ai risultati calcolati con il metodo di riferimento.

L'equalizzazione ottimizza la distribuzione dei livelli, non la qualità percepita: nelle zone che occupavano pochi livelli ingrandisce anche le piccole variazioni (rumore compreso) e può dare all'immagine un aspetto poco naturale, per cui non sempre migliora una fotografia.

DA RICORDARE

Un'operazione puntuale è una funzione dei soli livelli di grigio e si realizza con una LUT di 256 valori. L'istogramma conta i pixel per livello; lo stretching mappa linearmente il minimo su 0 e il massimo su 255; l'equalizzazione usa la somma cumulata dell'istogramma, divisa per il numero di pixel, per ottenere una distribuzione quasi uniforme. Per non alterare il risultato gli algoritmi scrivono sempre in una immagine di uscita distinta.

Rotazione, scaling e shear di immagini

Come applicare una trasformazione affine ai pixel di una immagine: mapping diretto e inverso, la classe PMatrix2D, replication e interpolazione bilineare.

Mapping diretto e mapping inverso

Nel capitolo 4 le trasformazioni cambiavano il sistema di coordinate del *disegno*. Qui si vogliono invece **ricalcolare i pixel** di una immagine, per ottenerne una nuova ruotata, ingrandita o inclinata. Sono i due modi del capitolo 12 della dispensa di teoria:

- **mapping diretto** (*forward*): si scorrono i pixel dell'*ingresso* e si calcola dove va a finire ciascuno nell'*uscita*;
- **mapping inverso** (*inverse*): si scorrono i pixel dell'*uscita* e per ciascuno si calcola, con la matrice inversa, da quale punto dell'*ingresso* proviene; il valore si ricava da quel punto, eventualmente per interpolazione.

Le coordinate calcolate sono numeri reali, e vanno ricondotte a pixel interi. Nel mapping diretto alcuni pixel di uscita non vengono mai colpiti e restano **vuoti**; con il mapping inverso ogni pixel di uscita è visitato una sola volta, per cui il metodo inverso è quello che si usa in pratica. Nel corso la teoria indica i pixel con (riga, colonna); in questo capitolo, come in tutta la dispensa di laboratorio, un pixel è $(x, y) = (\text{colonna}, \text{riga})$ e la matrice è quella di Processing (vettori colonna, capitolo 4): il punto di partenza (u, v) del pixel di uscita (x, y) si ottiene con la matrice inversa, $(u, v) = M^{-1} \cdot (x, y)$.

La classe PMatrix2D

Per costruire e invertire la matrice si usa la classe PMatrix2D, che è la stessa che Processing usa internamente per le trasformazioni del disegno. Si crea come identità con `new PMatrix2D()` e le si applicano `translate()`, `rotate()`, `scale()` con la stessa regola del capitolo 4: ogni chiamata moltiplica a destra, quindi l'ultima agisce per prima sui punti. I metodi che servono in questo capitolo sono:

Metodo	Effetto
<code>m.translate(tx, ty)</code>	componi con una traslazione
<code>m.rotate(a)</code> / <code>m.scale(sx, sy)</code>	componi con una rotazione (radianti) o con una scala
<code>m.apply(n00, n01, n02, n10, n11, n12)</code>	componi con una matrice affine qualsiasi (per righe)
<code>m.invert()</code>	sostituisce la matrice con la sua inversa (restituisce false se non è invertibile)
<code>m.mult(PVector p, PVector q)</code>	calcola in q l'immagine del punto p e la restituisce
<code>new PMatrix2D(m)</code>	crea una copia di m
campi <code>m00 ... m12</code>	i sei coefficienti della matrice (prime due righe, come <code>printMatrix()</code>)

Tabella 1. I metodi di PMatrix2D usati per le immagini.

ATTENZIONE ALLO SHEAR

Nella versione di Processing usata per queste prove i metodi `shearX()` e `shearY()` di `PMatrix2D` producono una matrice **non invertibile** (con `shearX(radians(30))` la matrice stampata risulta con `m11 = 0` e `m10 = 0,577` anziché `m01 = 0,577`, `m11 = 1`): i coefficienti passati internamente a `apply()` sono sbagliati. Le funzioni di disegno `shearX()` e `shearY()` sono invece corrette. Per lo shear di una immagine si usa quindi `apply()`, scrivendo la matrice a mano: per `shearX` con angolo a è `apply(1, tan(a), 0, 0, 1, 0)`.

Mapping inverso con replication e interpolazione bilineare

La funzione `trasforma()` crea una immagine di uscita $w \times h$ e, per ogni suo pixel (x, y) , calcola con la matrice **inversa** il punto (u, v) dell'ingresso da cui proviene; se cade fuori dall'immagine il pixel resta nero. Il valore di (u, v) , in generale non intero, si ricava in due modi:

- **replication** (`vicino()`): si prende il pixel più vicino, di coordinate $(\lfloor u + 0,5 \rfloor, \lfloor v + 0,5 \rfloor)$, cioè con l'arrotondamento di Processing;
- **interpolazione bilineare** (`bilineare()`): si media pesatamente il valore dei quattro pixel che circondano il punto. Con f_x e f_y le parti frazionarie di u e v , il valore è $(1 - f_y) \cdot [(1 - f_x) \cdot a + f_x \cdot b] + f_y \cdot [(1 - f_x) \cdot c + f_x \cdot d]$, dove a, b, c, d sono i quattro vicini (alto a sinistra, alto a destra, basso a sinistra, basso a destra).

Un dettaglio utile: con l'interpolazione bilineare la riga e la colonna più esterne non hanno quattro vicini, e per semplicità i pixel che escono da questo intervallo sono lasciati neri.

```
int vicino(PImage in, float u, float v) {           // replication: pixel piu' vicino
    int x = round(u), y = round(v);
    if (x < 0 || x >= in.width || y < 0 || y >= in.height) return color(0);
    return in.pixels[y * in.width + x];
}

int bilineare(PImage in, float u, float v) {         // media pesata dei 4 vicini
    int x0 = floor(u), y0 = floor(v);
    if (x0 < 0 || x0 + 1 >= in.width || y0 < 0 || y0 + 1 >= in.height) return color(0);
    float fx = u - x0, fy = v - y0;                // parti frazionarie
    float a = red(in.pixels[y0 * in.width + x0]);
    float b = red(in.pixels[y0 * in.width + x0 + 1]);
    float c = red(in.pixels[(y0 + 1) * in.width + x0]);
    float d = red(in.pixels[(y0 + 1) * in.width + x0 + 1]);
    float alto = a * (1 - fx) + b * fx;
    float basso = c * (1 - fx) + d * fx;
    return color(round(alto * (1 - fy) + basso * fy));
}

PImage trasforma(PImage in, PMatrix2D m, int w, int h, boolean liscia) {
    PMatrix2D inv = new PMatrix2D(m);
    inv.invert();                                   // inverse mapping: serve l'inversa
    PImage out = createImage(w, h, RGB);
    in.loadPixels();
    out.loadPixels();
    PVector p = new PVector();
    for (int y = 0; y < h; y++) {
        for (int x = 0; x < w; x++) {
            inv.mult(new PVector(x, y), p);         // da dove viene il pixel (x, y)?
            out.pixels[y * w + x] = liscia ? bilineare(in, p.x, p.y) : vicino(in, p.x, p.y);
        }
    }
    out.updatePixels();
    return out;
}
```

Rotazione

La rotazione di 30° attorno al **centro** dell'immagine (256, 256) si costruisce come nel capitolo 4: si porta il centro nell'origine, si ruota, si riporta il centro nella posizione di partenza. Nel codice si scrive in ordine inverso rispetto a come agisce sui punti: prima `translate(256, 256)`, poi `rotate()`, infine `translate(-256, -256)`.

```
void setup() {
  size(100, 100);
  PImage in = loadImage("camera.png");
  PMatrix2D m = new PMatrix2D();           // identita'
  m.translate(256, 256);                   // stessa logica di translate() ...
  m.rotate(radians(30));                  // ... e rotate() del capitolo 4
  m.translate(-256, -256);                 // rotazione attorno al centro
  trasforma(in, m, 512, 512, false).save("rot_vicino.png");
  trasforma(in, m, 512, 512, true).save("rot_bil.png");
  println(m.m00, m.m01, m.m02);
  println(m.m10, m.m11, m.m12);
}
```



Figura 1. Rotazione di 30° attorno al centro, con replication (a sinistra) e interpolazione bilineare (a destra); i triangoli neri sono le zone che non provengono dall'ingresso.

```
0.8660254 -0.5 162.2975
0.5 0.8660254 -93.702515
```

La console mostra la matrice composta: 0.8660254 e -0.5 sono $\cos 30^\circ = 0,866$ e $-\sin 30^\circ = -0,5$, e le due traslazioni (162.2975 e -93.702515) sono quelle che fanno ruotare l'immagine attorno al suo centro; coincidono con i valori calcolati a mano: $256 - 256 \cos 30^\circ + 256 \sin 30^\circ = 162,30$ e $256 - 256 \sin 30^\circ - 256 \cos 30^\circ = -93,70$. Il risultato con la replication è identico, pixel per pixel, a un calcolo di riferimento in doppia precisione; quello bilineare differisce da esso al più di un livello in 34 pixel su 262 144, per l'arrotondamento dei numeri float.

ROTAZIONE E RITORNO

Un buon collaudo consiste nel ruotare di un angolo e poi dell'angolo opposto, e nel confrontare il risultato con l'originale (nella sola zona centrale 300×300 , che non perde pixel ai bordi). Con 90° la trasformazione è esatta, perché i punti calcolati cadono su pixel interi; con 30° ogni rotazione introduce un arrotondamento, e il risultato dipende dal metodo.

```

PMatrix2D rotazione(float gradi) { // rotazione attorno al centro (256, 256)
    PMatrix2D m = new PMatrix2D();
    m.translate(256, 256);
    m.rotate(radians(gradi));
    m.translate(-256, -256);
    return m;
}

void setup() {
    size(100, 100);
    PImage in = loadImage("camera.png");
    PImage c0 = in.get(106, 106, 300, 300); // zona centrale, sempre dentro
    for (float g : new float[]{90, 30}) {
        for (int liscia = 0; liscia < 2; liscia++) {
            PImage a = trasforma(in, rotazione(g), 512, 512, liscia == 1);
            PImage b = trasforma(a, rotazione(-g), 512, 512, liscia == 1);
            PImage c1 = b.get(106, 106, 300, 300);
            println(g, liscia == 1 ? "bilineare" : "vicino", psnr(c0, c1));
        }
    }
}

```

```

90.0 vicino Infinity
90.0 bilineare Infinity
30.0 vicino 31.998236
30.0 bilineare 33.38313

```

Con 90° il PSNR è infinito (immagine identica) con entrambi i metodi. Con 30° la sequenza di due replication porta a 32,00 dB, mentre due interpolazioni bilineari consecutive danno 33,38 dB: l'interpolazione costa un po' di sfocatura ma evita gli errori di posizione di mezzo pixel della replication. I due valori sono stati confermati dal calcolo di riferimento (31,998 e 33,383 dB).

Scaling e shear

Per un ingrandimento si usa `scale(s)`, e per portare nella finestra di uscita una zona precisa si compone con una traslazione. Nell'esempio la zona di 80 × 80 pixel che inizia in (196, 72) (capelli e fronte) viene ingrandita 4 volte: la matrice è `scale(4)` seguita, nell'ordine di scrittura, da `translate(-196, -72)`, cioè prima si sposta la zona nell'origine e poi la si ingrandisce. Lo shear orizzontale di 30° sposta ogni punto di $\tan 30^\circ \cdot y = 0,577 \cdot y$ verso destra: l'immagine di uscita deve essere più larga ($512 + 512 \cdot 0,577 \approx 808$ pixel).

```

void setup() {
    size(100, 100);
    PImage in = loadImage("camera.png");
    PMatrix2D m = new PMatrix2D();
    m.scale(4); // ingrandimento 4x ...
    m.translate(-196, -72); // ... della zona in (196, 72)
    trasforma(in, m, 320, 320, false).save("zoom_vicino.png");
    trasforma(in, m, 320, 320, true).save("zoom_bil.png");

    PMatrix2D s = new PMatrix2D();
    s.apply(1, tan(radians(30)), 0, // x' = x + y * tan(30 gradi)
           0, 1, 0); // y' = y
    trasforma(in, s, 808, 512, true).save("shear.png");
    println(s.m00, s.m01, s.m10, s.m11);
}

```



Figura 2. Ingrandimento 4x di una zona di 80×80 pixel: la replication produce blocchi di 4×4 pixel, l'interpolazione bilineare sfuma i passaggi.



Figura 3. Shear orizzontale della fotografia con interpolazione bilineare; le zone nere sono fuori dall'immagine originale.

1.0 0.57735026 0.0 1.0

La console conferma i coefficienti: la matrice dello shear ha $m_{01} = 0,577$ (e $m_{11} = 1$). Le due immagini ingrandite coincidono con il calcolo di riferimento in tutti i pixel; lo shear con quello di riferimento tranne cinque pixel che differiscono di un livello, sempre per l'arrotondamento. Il confronto tra le due immagini ingrandite si può paragonare a quello della Figura 3 del capitolo 7 sulla replication: i bordi a gradini sono gli artefatti tipici del metodo del vicino più prossimo.

Perché il mapping diretto lascia dei buchi

Per mostrare perché non si usa il mapping diretto, lo sketch seguente ruota la fotografia di 30° scorrendo i pixel dell'ingresso e scrivendo ciascuno nella posizione calcolata (arrotondata). Poi conta i pixel di uscita che dovrebbero ricevere un valore (perché provengono da un punto dell'immagine) ma non sono mai stati scritti, e li colora di rosso.

```

void setup() {
    size(100, 100);
    PImage in = loadImage("camera.png");
    PMatrix2D m = new PMatrix2D();
    m.translate(256, 256);
    m.rotate(radians(30));
    m.translate(-256, -256);
    PMatrix2D inv = new PMatrix2D(m);
    inv.invert();

    PImage out = createImage(512, 512, RGB);           // tutto nero all'inizio
    boolean[] scritto = new boolean[512 * 512];
    in.loadPixels();
    out.loadPixels();
    PVector q = new PVector();
    for (int y = 0; y < 512; y++) {                     // si scorre l'ingresso
        for (int x = 0; x < 512; x++) {
            m.mult(new PVector(x, y), q);              // dove va il pixel (x, y)?
            int xi = round(q.x), yi = round(q.y);
            if (xi >= 0 && xi < 512 && yi >= 0 && yi < 512) {
                out.pixels[yi * 512 + xi] = in.pixels[y * 512 + x];
                scritto[yi * 512 + xi] = true;
            }
        }
    }
    int buchi = 0;                                     // pixel che dovrebbero avere un valore
    for (int y = 0; y < 512; y++) {
        for (int x = 0; x < 512; x++) {
            inv.mult(new PVector(x, y), q);
            boolean dentro = q.x >= 0 && q.x <= 511 && q.y >= 0 && q.y <= 511;
            if (dentro && !scritto[y * 512 + x]) {
                out.pixels[y * 512 + x] = color(255, 0, 0); // buco segnalato in rosso
                buchi++;
            }
        }
    }
    out.updatePixels();
    out.save("avanti.png");
    println("buchi:", buchi);
}

```

Mapping diretto, rotazione di 30°



Figura 4. Con il mapping diretto compaiono pixel vuoti (in rosso) sparsi in tutta l'immagine ruotata.

buchi: 29596

I buchi sono 29596 su circa 262 144 pixel, più del 11 per cento: il calcolo di riferimento in doppia precisione ne dà 29 600, e la piccola differenza dipende, di nuovo, dall'arrotondamento dei float. Si potrebbe riempirli a posteriori con la media dei vicini, ma è più semplice e più preciso usare il mapping inverso.

DA RICORDARE

Per trasformare una immagine si usa il **mapping inverso**: per ogni pixel di uscita si applica la matrice inversa e si legge l'ingresso con la replication o con l'interpolazione bilineare. La matrice si costruisce con `PMatrix2D` componendo `translate()`, `rotate()`, `scale()` o `apply()`, con l'ultima chiamata che agisce per prima. Per ruotare attorno al centro: `translate(cx, cy)`, `rotate(a)`, `translate(-cx, -cy)`.

Filtri di rango: mediano, massimo e minimo

I filtri non lineari basati sull'ordinamento dei pixel di una finestra: come si costruiscono e come si comportano sul rumore sale e pepe.

Che cosa sono i filtri di rango

Un **filtro di rango** sostituisce ogni pixel con un valore scelto tra quelli della sua **finestra**, per esempio i 9 pixel di un quadrato 3×3 centrato sul pixel. Si ordinano i valori della finestra dal più piccolo al più grande e si prende quello di un dato **rango** (posizione nell'ordinamento):

Filtro	Valore scelto	Con 9 valori ordinati
Minimo (erosione)	il più piccolo	$v[0]$
Mediano	quello al centro	$v[4]$
Massimo (dilatazione)	il più grande	$v[8]$

Tabella 1. I tre filtri di rango con finestra 3×3 .

Sono filtri **non lineari**: il risultato non è una somma pesata dei pixel (come nella convoluzione del capitolo seguente), perché dipende dall'ordinamento. Il filtro mediano è il più usato per eliminare il rumore **sale e pepe** (*impulsive noise*), costituito da pixel isolati completamente neri (pepe) o completamente bianchi (sale); la sua trattazione teorica è nel capitolo 23 della dispensa di teoria.

Il rumore sale e pepe

Per provare i filtri servono immagini con un rumore noto. La funzione `rumore()` danneggia ciascun pixel con probabilità p : con probabilità $p/2$ lo mette a 0 (pepe), con probabilità $p/2$ a 255 (sale). L'istruzione `randomSeed(1)` fissa il seme del generatore di numeri casuali, così il risultato è lo stesso a ogni esecuzione.

```
final int MINIMO = 0, MEDIANA = 1, MASSIMO = 2;

PImage rumore(PImage in, float p) {
    randomSeed(1);
    PImage out = in.get();
    out.loadPixels();
    for (int i = 0; i < out.pixels.length; i++) {
        float r = random(1);
        if (r < p / 2) out.pixels[i] = color(0);
        else if (r < p) out.pixels[i] = color(255);
    }
    out.updatePixels();
    return out;
}
```

Implementazione

La funzione `filtroRango()` costruisce la finestra di raggio r (lato $2r + 1$: raggio 1 per 3×3 , raggio 2 per 5×5) e per ogni pixel raccoglie i valori in un vettore, lo ordina con `sort()` e sceglie l'elemento giusto. Ai bordi, dove la finestra uscirebbe dall'immagine, si **ripetono i pixel del bordo**: le coordinate sono limitate dentro l'immagine con `constrain()`.

```
PImage filtroRango(PImage in, int raggio, int tipo) {
    int n = (2 * raggio + 1) * (2 * raggio + 1);           // pixel nella finestra
    PImage out = createImage(in.width, in.height, RGB);
    in.loadPixels();
    out.loadPixels();
    int[] v = new int[n];
    for (int y = 0; y < in.height; y++) {
        for (int x = 0; x < in.width; x++) {
            int k = 0;
            for (int dy = -raggio; dy <= raggio; dy++) {
                for (int dx = -raggio; dx <= raggio; dx++) {
                    int xx = constrain(x + dx, 0, in.width - 1); // ai bordi si ripete il bordo
                    int yy = constrain(y + dy, 0, in.height - 1);
                    v[k++] = (int) red(in.pixels[yy * in.width + xx]);
                }
            }
            v = sort(v); // ordina i valori della finestra
            int scelto = (tipo == MINIMO) ? v[0] : (tipo == MASSIMO) ? v[n - 1] : v[n / 2];
            out.pixels[y * in.width + x] = color(scelto);
        }
    }
    out.updatePixels();
    return out;
}
```

Si applicano poi i tre filtri alla fotografia con il 10 per cento di pixel danneggiati, più la mediana con finestra 5×5 , e si calcola il PSNR rispetto alla fotografia originale (funzione del capitolo 7):

```
void setup() {
    size(100, 100);
    PImage a = loadImage("camera.png");
    PImage r = rumore(a, 0.10); // 10% dei pixel danneggiati
    r.save("rumore.png");
    PImage mn = filtroRango(r, 1, MINIMO);
    PImage mx = filtroRango(r, 1, MASSIMO);
    PImage md = filtroRango(r, 1, MEDIANA);
    PImage md5 = filtroRango(r, 2, MEDIANA);
    mn.save("min.png");
    mx.save("max.png");
    md.save("med3.png");
    md5.save("med5.png");
    println("rumore", psnr(a, r));
    println("minimo 3x3", psnr(a, mn));
    println("massimo 3x3", psnr(a, mx));
    println("mediana 3x3", psnr(a, md));
    println("mediana 5x5", psnr(a, md5));
}
```

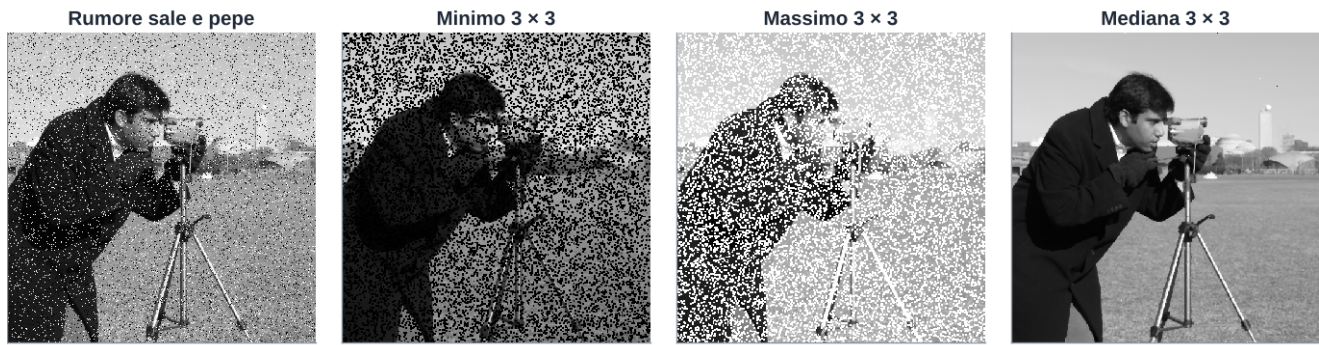


Figura 1. La fotografia con il 10 per cento di pixel danneggiati e il risultato dei tre filtri con finestra 3×3 : il minimo e il massimo peggiorano l'immagine, la mediana la ripulisce.

```
rumore 14.731711
minimo 3x3 8.862143
massimo 3x3 8.9853325
mediana 3x3 29.488373
mediana 5x5 27.592323
```

Il risultato è tutt'altro che banale. Il filtro **minimo** elimina i pixel bianchi (sale), perché il più piccolo valore della finestra non è mai un 255 isolato, ma allarga i neri: ogni pixel nero (pepe) invade tutta la finestra 3×3 e diventa un quadrato nero 3×3 . Il filtro **massimo** fa il contrario. Ne risultano PSNR molto bassi, 8,86 dB e 8,99 dB, **peggiori** di quello dell'immagine rumorosa (14,73 dB). Il filtro **mediano** elimina entrambi i tipi di rumore perché i valori estremi, essendo isolati, finiscono alle estremità dell'ordinamento e non vengono mai scelti: il PSNR sale a 29,49 dB.

Tutte e quattro le immagini filtrate sono state confrontate con i filtri di libreria (con lo stesso trattamento dei bordi) e coincidono in ogni pixel.

Perché la mediana funziona: un esempio

Lo sketch seguente isola una finestra 3×3 centrata nel pixel (154, 250) della fotografia rumorosa, dove il pixel centrale è un «sale»: elenca i nove valori per righe, li ordina e stampa minimo, mediana e massimo. (Contiene anche la funzione `rumore()` dello sketch precedente.)

```
void setup() {
  size(100, 100);
  PImage r = rumore(loadImage("camera.png"), 0.10);
  r.loadPixels();
  int x = 154, y = 250; // pixel centrale della finestra 3 x 3
  int[] v = new int[9];
  int k = 0;
  for (int dy = -1; dy <= 1; dy++)
    for (int dx = -1; dx <= 1; dx++)
      v[k++] = (int) red(r.pixels[(y + dy) * r.width + x + dx]);
  println("finestra:", join(nf(v, 0), " "));
  v = sort(v);
  println("ordinata:", join(nf(v, 0), " "));
  println("minimo", v[0], "mediana", v[4], "massimo", v[8]);
}
```

finestra: 27 28 30 27 255 28 28 29 28
ordinata: 27 27 28 28 28 28 29 30 255
minimo 27 mediana 28 massimo 255

Nella finestra otto valori stanno tra 27 e 30, e il pixel centrale vale 255. Il filtro minimo darebbe 27, il massimo darebbe 255 (il rumore resta), la mediana dà **28**: il valore tipico dell'intorno. Un valore anomalo non può mai spostare la mediana più di quanto la spostino i suoi vicini: è questo che rende il filtro robusto al rumore impulsivo, e distingue la mediana dalla **media** (la media dei nove valori sarebbe 53,3, molto lontana da 28).

Dimensione della finestra

Una finestra più grande elimina rumore più denso, ma sfuoca di più i dettagli sottili e arrotonda gli spigoli. Il confronto sulla fotografia mostra che la finestra 5×5 è più «pulita» ma meno nitida:



Figura 2. Mediana con finestra 3×3 e 5×5 sulla stessa fotografia rumorosa.

Con rumore al 10 per cento la finestra 3×3 ha il PSNR più alto (29,49 dB contro 27,59 dB della 5×5): per questa densità di rumore non serve di più, e le finestre grandi fanno solo perdere dettagli. Con rumore più denso, dove una finestra 3×3 contiene spesso più di 4 pixel danneggiati, la mediana 3×3 non basta più.

DA RICORDARE

Un filtro di rango ordina i pixel della finestra e sceglie il valore di un dato rango: minimo (rango 0), mediana (rango centrale), massimo (ultimo). Sono filtri non lineari. La mediana elimina il rumore sale e pepe senza sfocare gli spigoli come farebbe una media; minimo e massimo allargano rispettivamente le zone scure e quelle chiare. Ai bordi si ripetono i pixel del bordo.

Convoluzione: media N box, Sobel e Prewitt

Il filtraggio lineare con un kernel: come si implementa, il filtro di media N box e i filtri di Sobel e Prewitt per l'estrazione dei lati.

Il filtraggio con un kernel

Un **filtro lineare** calcola ogni pixel di uscita come somma pesata dei pixel di un intorno, con pesi dati da una piccola matrice detta **kernel** (o maschera). Con un kernel h di dimensione $(2r + 1) \times (2c + 1)$ centrato nell'origine, il pixel (x, y) dell'uscita è

$$g(x, y) = \sum_j \sum_i h(j, i) \cdot f(x + i, y + j)$$

con i da $-c$ a c e j da $-r$ a r , dove $h(j, i)$ è l'elemento della riga j e colonna i del kernel (con il centro in $(0, 0)$). Questa è la formula usata nelle lezioni del corso, che a rigore è una **correlazione**: la convoluzione vera ruota il kernel di 180° prima di applicarlo (capitolo 26 della teoria). Per i kernel simmetrici, come la media, le due operazioni sono identiche; per quelli antisimmetrici, come Sobel e Prewitt, la convoluzione vera cambia il segno della risposta, ma non il suo modulo. Nel codice seguente si implementa la formula del corso, e nel resto della dispensa si userà il nome `correlazione()`.

Il kernel è un array bidimensionale di `float` indicizzato `h[j][i]` (riga, colonna). Per mantenere la risposta senza arrotondamenti e senza limitare a $0 \dots 255$ (che per i filtri derivativi sarebbe sbagliato, perché la risposta può essere negativa o molto grande) la funzione restituisce un vettore di `float`. Per visualizzarla servirà `aImmagine()`. Ai bordi si **ripetono i pixel del bordo** (una delle quattro scelte del capitolo di teoria, che prevede anche zeri, toroidale o uso del solo centro).

```
float[] correlazione(PImage in, float[][] h) {           // risposta del filtro, senza limiti
    int rr = h.length / 2, rc = h[0].length / 2;      // raggi verticale e orizzontale
    float[] out = new float[in.pixels.length];
    in.loadPixels();
    for (int y = 0; y < in.height; y++) {
        for (int x = 0; x < in.width; x++) {
            float somma = 0;
            for (int j = -rr; j <= rr; j++) {
                for (int i = -rc; i <= rc; i++) {
                    int xx = constrain(x + i, 0, in.width - 1); // bordi ripetuti
                    int yy = constrain(y + j, 0, in.height - 1);
                    somma += h[j + rr][i + rc] * red(in.pixels[yy * in.width + xx]);
                }
            }
            out[y * in.width + x] = somma;
        }
    }
    return out;
}
```

La funzione seguente converte un vettore di risposte in una immagine, secondo la formula $p = \text{limite}(\text{arrotondamento}(v \cdot \text{scala} + \text{offset}), 0, 255)$: la scala e l'offset servono a rendere visibili anche risposte negative o molto piccole.

```
PImage aImmagine(float[] v, int w, int h, float scala, float offset) {
    PImage out = createImage(w, h, RGB);
    out.loadPixels();
    for (int i = 0; i < v.length; i++)
        out.pixels[i] = color(constrain(round(v[i] * scala + offset), 0, 255));
    out.updatePixels();
    return out;
}
```

Il filtro di media N box

Il filtro di **media N box** sostituisce ogni pixel con la media dei $N \times N$ pixel dell'intorno: il kernel ha tutti i pesi uguali a $1 / N^2$, in modo che la somma dei pesi sia 1 e la luminosità media dell'immagine non cambi. È un filtro **passa basso**: attenua il rumore ma sfuoca i dettagli, tanto più quanto più grande è N (capitolo 23 della teoria). La funzione `box(n)` costruisce il kernel.

```
float[][] box(int n) { // media: n x n pesi uguali a 1 / n^2
    float[][] h = new float[n][n];
    for (int j = 0; j < n; j++)
        for (int i = 0; i < n; i++) h[j][i] = 1.0 / (n * n);
    return h;
}

float[][] SOBEL_X = {{-1, -2, -1}, { 0, 0, 0}, {1, 2, 1}}; // lati orizzontali
float[][] SOBEL_Y = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}}; // lati verticali
float[][] PREWITT_X = {{-1, -1, -1}, {0, 0, 0}, {1, 1, 1}};
float[][] PREWITT_Y = {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};
```

Si applica il filtro con $N = 3$ e $N = 7$ alla fotografia, e si valuta il PSNR rispetto all'originale. Il listato seguente contiene anche le prove sui filtri di Sobel e Prewitt, descritte nei paragrafi successivi.

```

void setup() {
    size(100, 100);
    PImage a = loadImage("camera.png");
    int w = a.width, h = a.height;
    for (int n : new int[]{3, 7}) {
        PImage b = aImmagine(correlazione(a, box(n)), w, h, 1, 0);
        b.save("box" + n + ".png");
        println("media", n, "x", n, psnr(a, b));
    }
    float[] sx = correlazione(a, SOBEL_X), sy = correlazione(a, SOBEL_Y);
    float[] px = correlazione(a, PREWITT_X), py = correlazione(a, PREWITT_Y);
    float[] ms = new float[w * h], mp = new float[w * h];
    for (int i = 0; i < ms.length; i++) {
        ms[i] = sqrt(sx[i] * sx[i] + sy[i] * sy[i]);          // modulo del gradiente
        mp[i] = sqrt(px[i] * px[i] + py[i] * py[i]);
    }
    aImmagine(sx, w, h, 0.25, 128).save("sobel_x.png");    // 128 = nessuna risposta
    aImmagine(sy, w, h, 0.25, 128).save("sobel_y.png");
    aImmagine(ms, w, h, 0.5, 0).save("sobel_mod.png");
    aImmagine(mp, w, h, 2.0 / 3, 0).save("prewitt_mod.png");
    PImage lati = createImage(w, h, RGB);                  // soglia 200 sul modulo di Sobel
    lati.loadPixels();
    for (int i = 0; i < ms.length; i++)
        lati.pixels[i] = ms[i] >= 200 ? color(0) : color(255);
    lati.updatePixels();
    lati.save("lati.png");
}

```

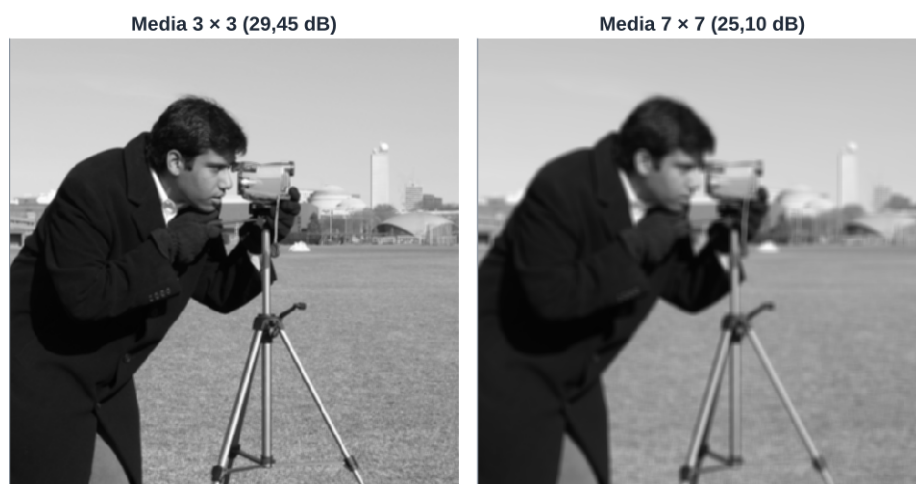


Figura 1. La fotografia filtrata con la media N box: più grande è la finestra, più forte è la sfocatura e più basso il PSNR rispetto all'originale.

```

media 3 x 3 29.449184
media 7 x 7 25.098036

```

Le due immagini filtrate sono identiche, pixel per pixel, a quelle calcolate con il metodo di riferimento (stessa gestione dei bordi). Il PSNR scende da 29,45 dB per la finestra 3×3 a 25,10 dB per quella 7×7 : un filtro passa basso toglie le alte frequenze, che sono parte del contenuto dell'immagine, e il confronto con l'originale lo penalizza. Per rimuovere il rumore il filtro di media è utile solo se il rumore è più forte di questa perdita.

Lati: i kernel di Sobel e Prewitt

I **lati** (*edge*) sono i punti in cui la luminosità cambia bruscamente; per trovarli si approssima la derivata prima dell'immagine con una differenza tra pixel opposti (capitolo 24 della teoria). I filtri di Prewitt e di Sobel calcolano la differenza tra le due righe (o colonne) ai lati del pixel, e allo stesso tempo mediano nella direzione ortogonale per ridurre il rumore: in modo uniforme per Prewitt, con pesi 1, 2, 1 per Sobel.

Sobel x	Prewitt x	Sobel y	Prewitt y																																				
<table><tr><td>-1</td><td>-2</td><td>-1</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>2</td><td>1</td></tr></table>	-1	-2	-1	0	0	0	1	2	1	<table><tr><td>-1</td><td>-1</td><td>-1</td></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	-1	-1	-1	0	0	0	1	1	1	<table><tr><td>-1</td><td>0</td><td>1</td></tr><tr><td>-2</td><td>0</td><td>2</td></tr><tr><td>-1</td><td>0</td><td>1</td></tr></table>	-1	0	1	-2	0	2	-1	0	1	<table><tr><td>-1</td><td>0</td><td>1</td></tr><tr><td>-1</td><td>0</td><td>1</td></tr><tr><td>-1</td><td>0</td><td>1</td></tr></table>	-1	0	1	-1	0	1	-1	0	1
-1	-2	-1																																					
0	0	0																																					
1	2	1																																					
-1	-1	-1																																					
0	0	0																																					
1	1	1																																					
-1	0	1																																					
-2	0	2																																					
-1	0	1																																					
-1	0	1																																					
-1	0	1																																					
-1	0	1																																					

Figura 2. I kernel di Sobel e di Prewitt con la denominazione del corso: il filtro «x» risponde ai lati orizzontali, il filtro «y» a quelli verticali.

La denominazione è quella delle slide del corso (e può essere opposta in altri testi): il filtro **x** calcola la differenza tra la riga sotto e la riga sopra, quindi risponde ai **lati orizzontali**; il filtro **y** calcola la differenza tra la colonna a destra e quella a sinistra e risponde ai **lati verticali**. Il kernel si scrive in Processing come array di array: nell'esempio ogni riga di graffe è una riga del kernel.

Il collaudo più semplice è sull'immagine di 6 × 6 pixel con un lato verticale, 10 a sinistra e 100 a destra: il filtro per i lati verticali deve rispondere con 4 · 90 = 360 (la somma dei pesi positivi di Sobel è 4) nelle due colonne a cavallo del lato, e il filtro per i lati orizzontali con 0 ovunque.

```
void setup() {
  size(100, 100);
  PImage f = createImage(6, 6, RGB);
  f.loadPixels();
  for (int y = 0; y < 6; y++)
    for (int x = 0; x < 6; x++) f.pixels[y * 6 + x] = color(x < 3 ? 10 : 100);
  float[] gx = correlazione(f, SOBEL_X);
  float[] gy = correlazione(f, SOBEL_Y);
  for (int x = 0; x < 6; x++) print(round(gx[2 * 6 + x]), "");
  println("  Sobel x, riga 2");
  for (int x = 0; x < 6; x++) print(round(gy[2 * 6 + x]), "");
  println("  Sobel y, riga 2");
}
```

```
0 0 0 0 0 0    Sobel x, riga 2
0 0 360 360 0 0  Sobel y, riga 2
```

Lo sketch stampa esattamente la risposta attesa, come nell'esempio del capitolo di teoria: 360 nelle colonne 2 e 3 per il filtro dei lati verticali e zero dappertutto per l'altro.

MODULO DEL GRADIENTE E SOGLIA

Le due risposte g_x e g_y si combinano nel **modulo del gradiente**

$$M = \sqrt{g_x^2 + g_y^2}$$

che è grande nei pixel «di lato» qualunque sia la direzione del lato. Fissando una **soglia** si ottiene una mappa binaria dei lati. Lo sketch (la parte finale del listato precedente) salva le risposte di Sobel e il modulo, per Sobel e per Prewitt, e la mappa dei lati con soglia 200 sul modulo di Sobel. Per la visualizzazione la risposta con segno viene scalata di un quarto e portata attorno a 128 (il grigio medio rappresenta risposta nulla), mentre il modulo è scalato di un mezzo per Sobel e di due terzi per Prewitt (che pesa i lati netti 3 invece di 4).

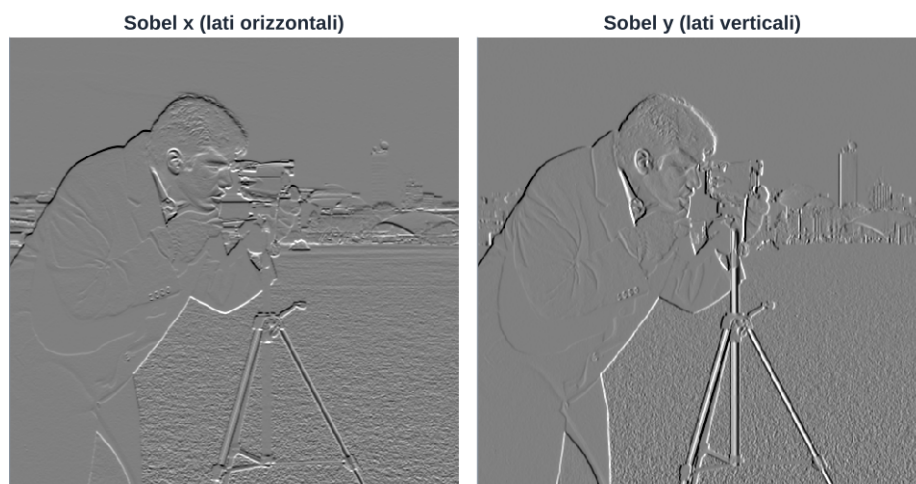


Figura 3. Risposta dei due filtri di Sobel, scalata di un quarto e centrata su 128: i valori chiari e scuri indicano lati in un senso o nell'altro.



Figura 4. Il modulo di Sobel e di Prewitt (scalati per la visualizzazione) e la mappa binaria dei lati ottenuta con soglia 200 sul modulo di Sobel: i pixel neri sono i pixel di lato.

Tutte le immagini ottenute (le due risposte di Sobel, i due moduli e la mappa dei lati) sono state confrontate con il calcolo di riferimento e coincidono in ogni pixel. Con la soglia 200 risultano di lato il 5,04 per cento dei pixel. Come osservato nel capitolo di teoria, il modulo di Sobel e quello di Prewitt sono molto simili. Una soglia bassa lascia anche tessitura e rumore (come nel prato in basso), una soglia alta spezza i contorni: la scelta dipende dall'applicazione.

DA RICORDARE

Un filtro lineare somma i prodotti tra kernel e pixel vicini: con la formula del corso è una correlazione, uguale alla convoluzione per i kernel simmetrici. La media N box ha pesi $1/N^2$ e sfuoca; Sobel e Prewitt approssimano la derivata prima e si combinano nel modulo, poi si applica una soglia. Il risultato va tenuto in float senza limitarlo, finché non serve visualizzarlo, scalandolo e centrandolo su 128 se ha segno.

Estrazione dei bit planes

Scomporre una immagine in piani di bit con le operazioni sui bit, ricostruirla con pochi piani e confrontare il binario puro con il codice Gray.

Piani di bit

Un livello di grigio a 8 bit è un numero binario di otto cifre, e si può scrivere come somma di potenze di 2 (capitolo 21 della dispensa di teoria):

$$f = \sum_{k=0...7} b_k \cdot 2^k$$

Il **piano di bit** k di una immagine è l'immagine binaria formata dal bit b_k di tutti i pixel. Il piano 7 è il **più significativo** (pesa 128), il piano 0 il **meno significativo** (pesa 1). In Processing il bit k di un intero v si estrae con due operazioni: lo spostamento a destra di k posti, $v \gg k$, che porta il bit nella posizione 0, e la maschera $\& 1$ che azzerà tutti gli altri.

Operatore	Significato	Esempio
>>	sposta i bit a destra	210 >> 4 è 13
&	AND bit a bit	210 & 15 è 2
^	XOR (OR esclusivo) bit a bit	12 ^ 10 è 6
binary(v, 8)	stringa binaria di 8 cifre	binary(210, 8) è 11010010

Tabella 1. Le operazioni sui bit usate in questo capitolo.

L'esempio lavora sul pixel (100, 50) della fotografia, che vale 210. Nella console compare la sua rappresentazione binaria 11010010: il piano 7 vale 1, il 6 vale 1, il 5 vale 0, il 4 vale 1, e così via, partendo dalla cifra più a sinistra.

```

PImage piano(PImage in, int k) {
    // piano k (0 = meno significativo)
    PImage out = createImage(in.width, in.height, RGB);
    in.loadPixels();
    out.loadPixels();
    for (int i = 0; i < in.pixels.length; i++) {
        int v = (int) red(in.pixels[i]);
        int bit = (v >> k) & 1; // sposta il bit in fondo e isolalo
        out.pixels[i] = bit == 1 ? color(255) : color(0);
    }
    out.updatePixels();
    return out;
}

PImage tieniPiani(PImage in, int primo) {
    // conserva solo i piani da 'primo' a 7
    PImage out = createImage(in.width, in.height, RGB);
    in.loadPixels();
    out.loadPixels();
    int maschera = (0xFF << primo) & 0xFF; // es. primo = 4 -> 11110000
    for (int i = 0; i < in.pixels.length; i++)
        out.pixels[i] = color((int) red(in.pixels[i]) & maschera);
    out.updatePixels();
    return out;
}

```

```

PImage codiceGray(PImage in) {
    // g = v XOR (v >> 1)
    PImage out = createImage(in.width, in.height, RGB);
    in.loadPixels();
    out.loadPixels();
    for (int i = 0; i < in.pixels.length; i++) {
        int v = (int) red(in.pixels[i]);
        out.pixels[i] = color(v ^ (v >> 1));
    }
    out.updatePixels();
    return out;
}

```

I piani della fotografia

Lo sketch seguente salva gli otto piani di bit della fotografia (i bit a 1 diventano bianchi, quelli a 0 neri), quattro piani del codice Gray (descritto più avanti) e le ricostruzioni con pochi piani.

```

void setup() {
    size(100, 100);
    PImage a = loadImage("camera.png");
    int v = (int) red(a.get(100, 50));
    println(v, binary(v, 8));
    for (int k = 0; k < 8; k++) piano(a, k).save("p" + k + ".png");
    for (int k = 7; k >= 4; k--) piano(codiceGray(a), k).save("g" + k + ".png");
    for (int primo = 7; primo >= 4; primo--) {
        // quanti piani servono?
        PImage r = tieniPiani(a, primo);
        r.save("t" + primo + ".png");
        println("piani", primo, "...7", psnr(a, r));
    }
    println(binary(127, 8), binary(128, 8));
    println(binary(127 ^ (127 >> 1), 8), binary(128 ^ (128 >> 1), 8)); // Gray
}

```

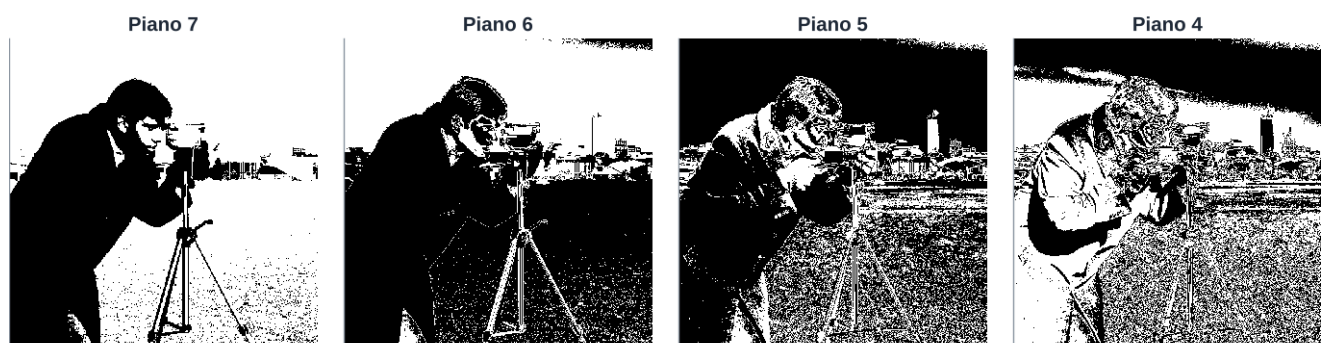


Figura 1. I quattro piani più significativi della fotografia: il piano 7 coincide con una soglia a 128, e ogni piano successivo aggiunge dettaglio.

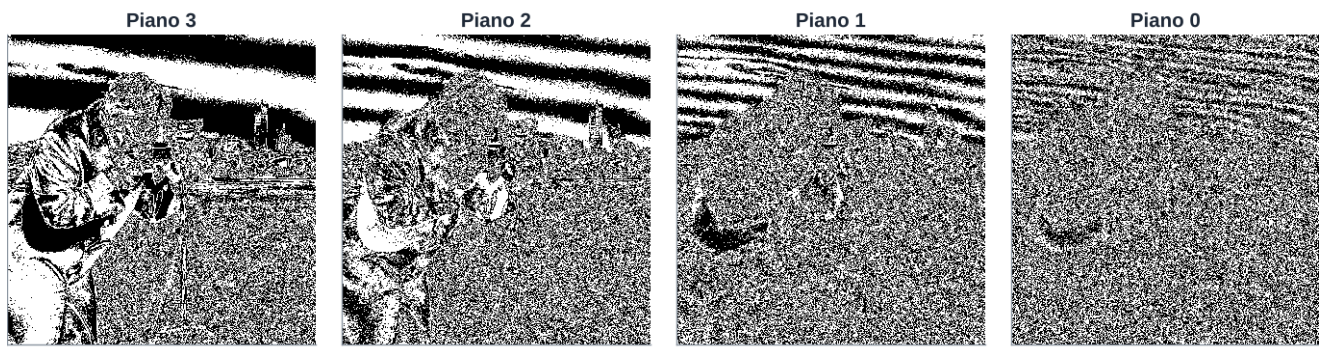


Figura 2. I quattro piani meno significativi: sono praticamente rumore e contribuiscono pochissimo al valore del pixel (al più 8, 4, 2 e 1 livelli).

Tutti gli otto piani sono stati confrontati con il calcolo di riferimento e coincidono. Il piano 7 è un'immagine in bianco e nero che riproduce già la struttura generale della scena, perché il bit più significativo dice se il pixel è chiaro (da 128 a 255) o scuro (da 0 a 127); i piani 6, 5 e 4 aggiungono i mezzi toni. Nei piani più bassi i bit variano da un pixel all'altro come in un rumore.

Ricostruire l'immagine con pochi piani

Se si tengono solo i piani da primo a 7 e si azzerano gli altri, si ottiene una immagine con 2^n livelli, dove $n = 8 - \text{primo}$ è il numero di piani conservati. La funzione `tieniPiani()` lo fa con una **maschera**, per esempio 11110000 per tenere gli ultimi quattro piani, e un AND bit a bit con il pixel. È la quantizzazione uniforme del capitolo 7, ma con i valori riportati al **limite inferiore** dell'intervallo e non al centro (per questo risultano tutti un po' troppo bassi).

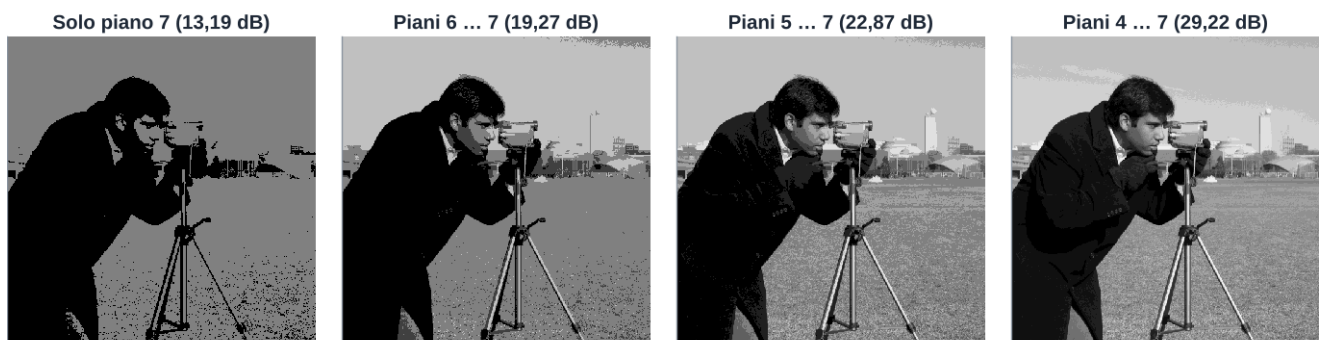


Figura 3. Ricostruzione con 1, 2, 3 e 4 piani (i più significativi) e PSNR rispetto all'originale: già con 4 piani l'immagine è riconoscibile, con falsi contorni nelle zone sfumate.

```
210 11010010
piani 7 ... 7 13.19197
piani 6 ... 7 19.269583
piani 5 ... 7 22.869047
piani 4 ... 7 29.216028
01111111 10000000
01000000 11000000
```

Si osserva che il PSNR cresce con i piani aggiunti: 13,19 dB con un solo piano, 19,27 con due, 22,87 con tre, 29,22 con quattro. Aggiungere un piano raddoppia il numero di livelli, ma il guadagno non è costante (qui da 3,6 a 6,4 dB per piano): il vantaggio dei piani alti è netto, mentre i piani 3, 2, 1 e 0 contribuiscono con lievi correzioni e sono i primi che si scartano nella compressione. Nel capitolo 21 della teoria si nota inoltre che azzerare un piano **non elimina un intervallo di toni**: sottrae la potenza di 2 corrispondente solo ai pixel che hanno quel bit a 1.

Il codice Gray

Il binario puro ha un difetto: due livelli consecutivi possono differire in molti bit (127 è 01111111 e 128 è 10000000: cambiano tutti e otto), e quindi un piccolo cambiamento di luminosità può far cambiare moltissimi piani. Nel **codice Gray** i valori consecutivi differiscono sempre di un solo bit, e i piani sono più regolari. Il codice Gray g di un valore v si calcola con uno XOR e uno spostamento:

$$g = v \text{ XOR } (v \text{ spostato a destra di } 1) \quad \text{in Processing: } v \wedge (v \gg 1)$$

L'ultima parte dell'esempio stampa il codice binario e quello Gray di 127 e 128. La funzione `codiceGray()`, nel primo listato, converte l'intera immagine; i piani del Gray si estraggono come quelli di una immagine qualsiasi, con la funzione `piano()`.

Le ultime due righe della console mostrano il binario puro di 127 e 128 e, sotto, il loro codice Gray:

```
01111111 10000000
01000000 11000000
```

Con il binario puro 127 e 128 differiscono in tutti gli otto bit; con il Gray in uno solo (il bit 7: da 01000000 a 11000000).

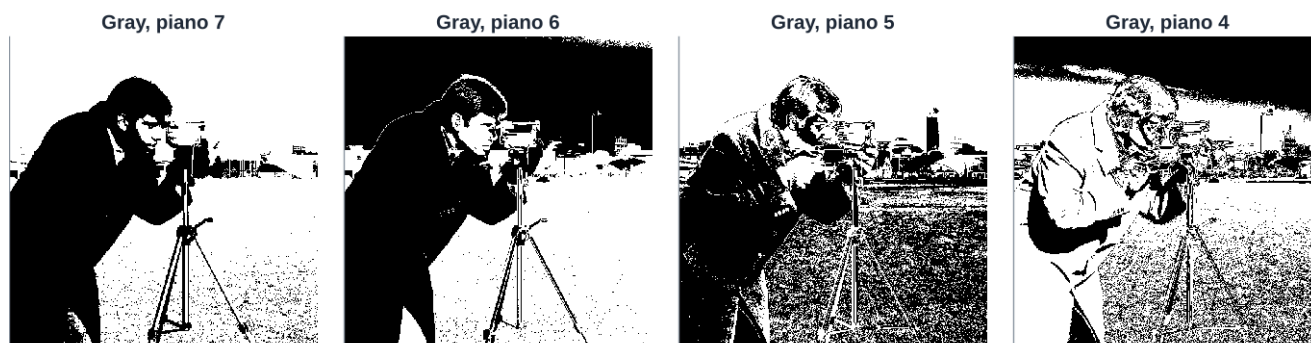


Figura 4. I quattro piani più significativi del codice Gray: il piano 7 è identico a quello del binario puro, i successivi sono più uniformi nelle zone di texture, come il prato.

Il piano 7 del Gray è uguale a quello del binario, perché lo XOR lascia invariato il bit più alto. I piani successivi sono la differenza (XOR) di due piani adiacenti del binario. L'effetto si vede bene nel prato, dove i livelli sono vicini alle soglie dei piani: nel binario puro i bit alti si ribaltano per variazioni minime e i piani risultano granulosi, nel Gray cambia un solo bit alla volta e i piani sono molto più uniformi.

TERMINOLOGIA

«Codice Gray» (Frank Gray) è il nome di questa codifica binaria, e non significa «scala di grigio»: sono due cose diverse, anche se la parola inglese *gray* è la stessa. Anche i piani del codice Gray si possono rappresentare come immagini in bianco e nero.

DA RICORDARE

Il piano k è $(v \gg k) \& 1$, mostrato con 255 per i bit a 1. Il piano 7 pesa 128 e il piano 0 pesa 1. Con $v \& \text{maschera}$ si tengono i piani da primo a 7. Il codice Gray, $v \wedge (v \gg 1)$, fa cambiare un solo bit tra livelli consecutivi.