

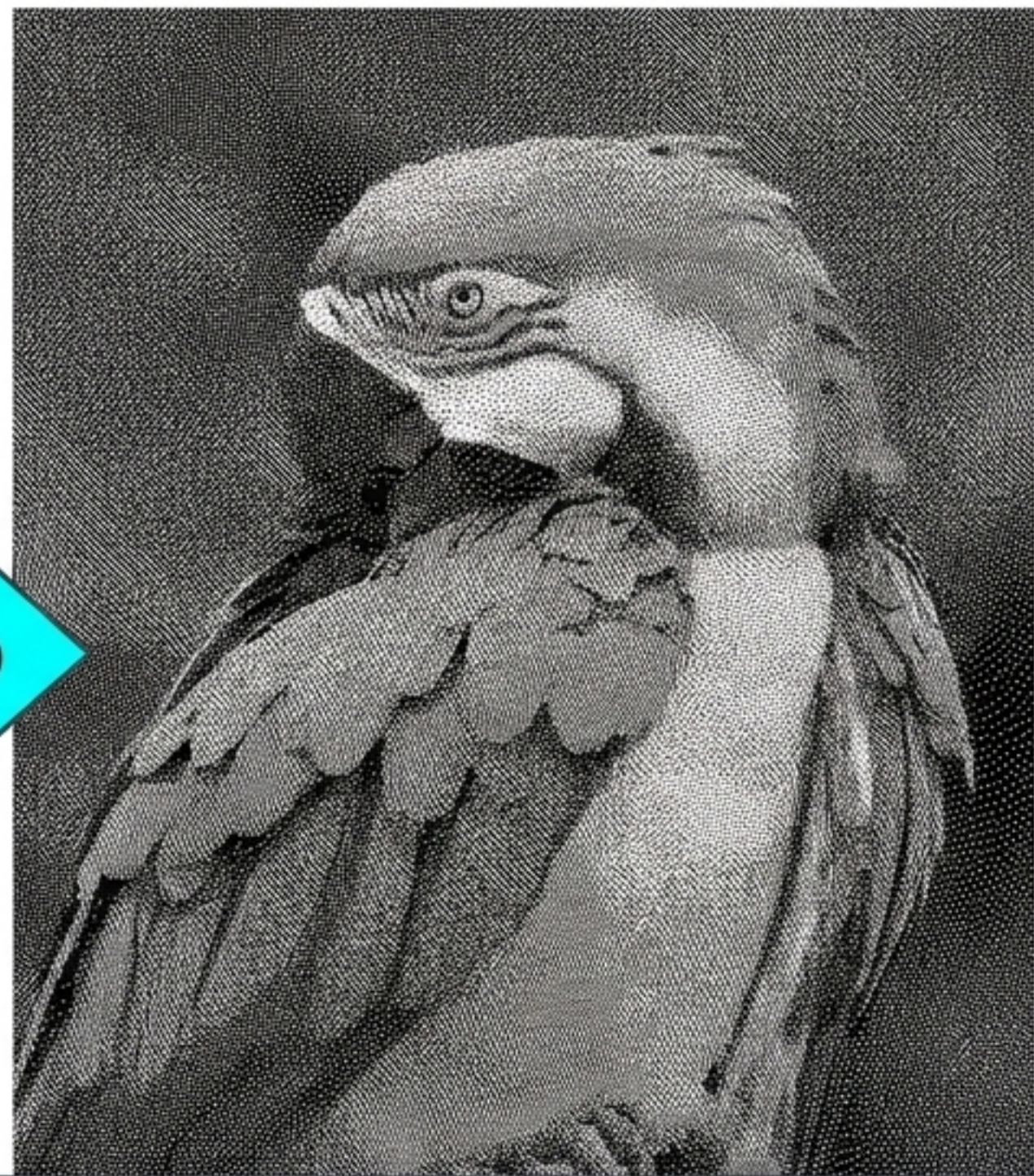
誤差拡散法の実装と解剖

Pythonコードで読み解く、2値化アルゴリズムの内部構造

128	255	-14	49	200	3
128	255	-97	-14	9	3
-90	-90	12	75	156	3
-22	10	44	-5	101	101



`error_diffusion_2bit()`



入力：グレースケール画像（8-bit）

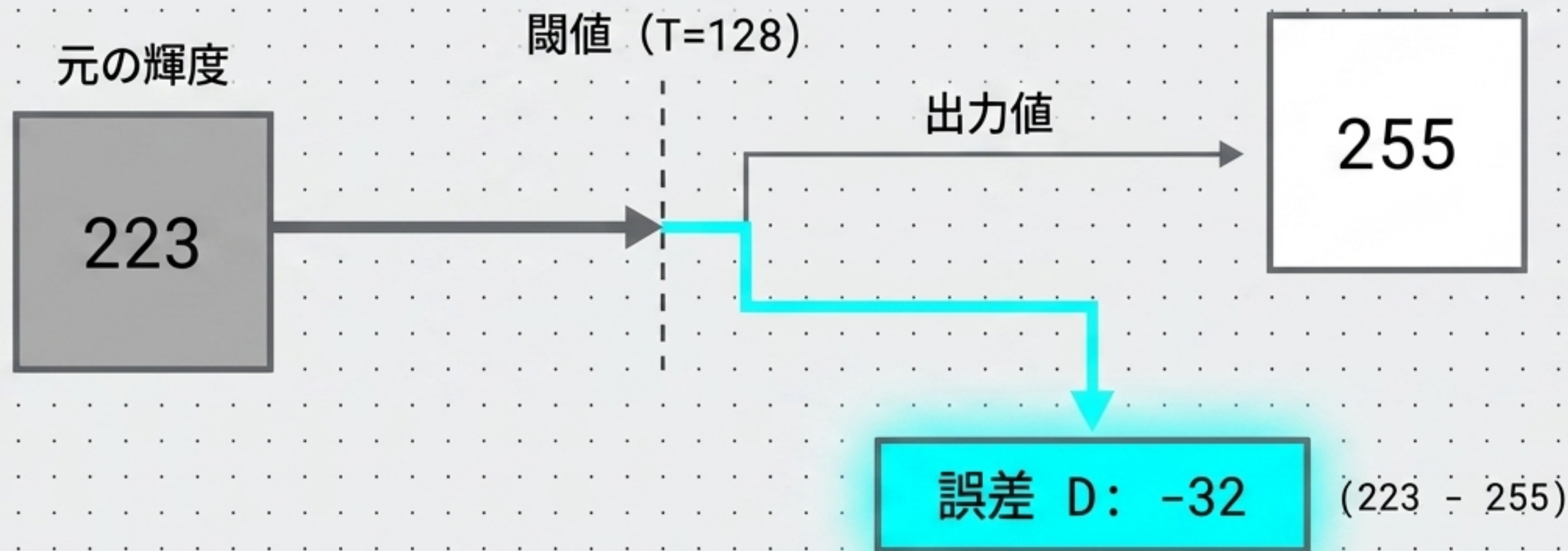
0～255の連続した輝度値を持つピクセルの集合。

出力：モノクロ2値画像（1-bit）

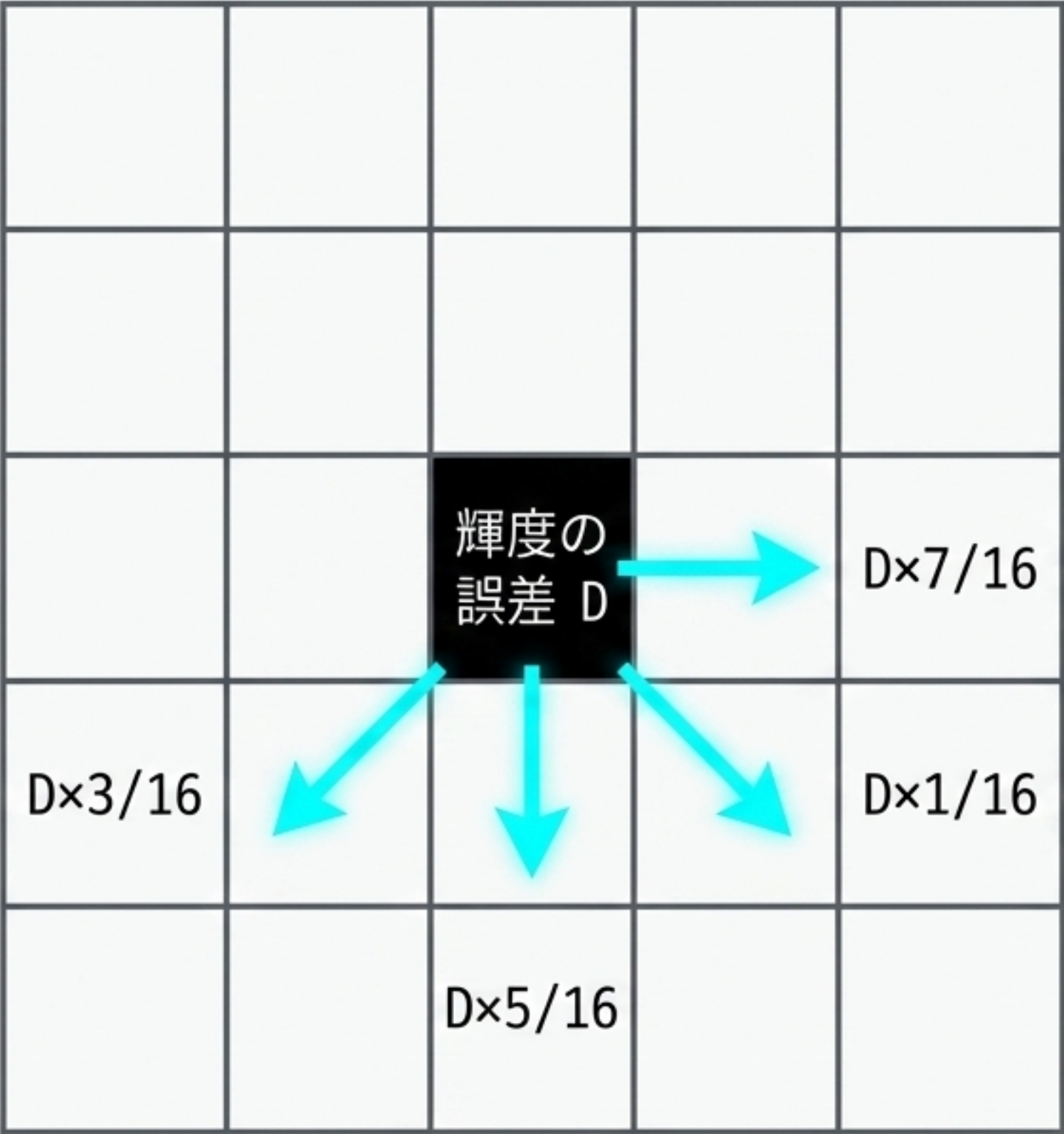
白(255)と黒(0)のドットのみ。密度の違いによって、人間の目に豊かな階調（グレー）を「錯覚」させる。

誤差 (Error D) の正体

ピクセルを白(255)か黒(0)に強制変換した際、元の輝度との間に生じる「ズレ」が誤差。
この失われた情報を捨てず、隣接する未処理ピクセルに押し付ける（拡散する）ことで、
画像全体の明るさの総量を維持する。



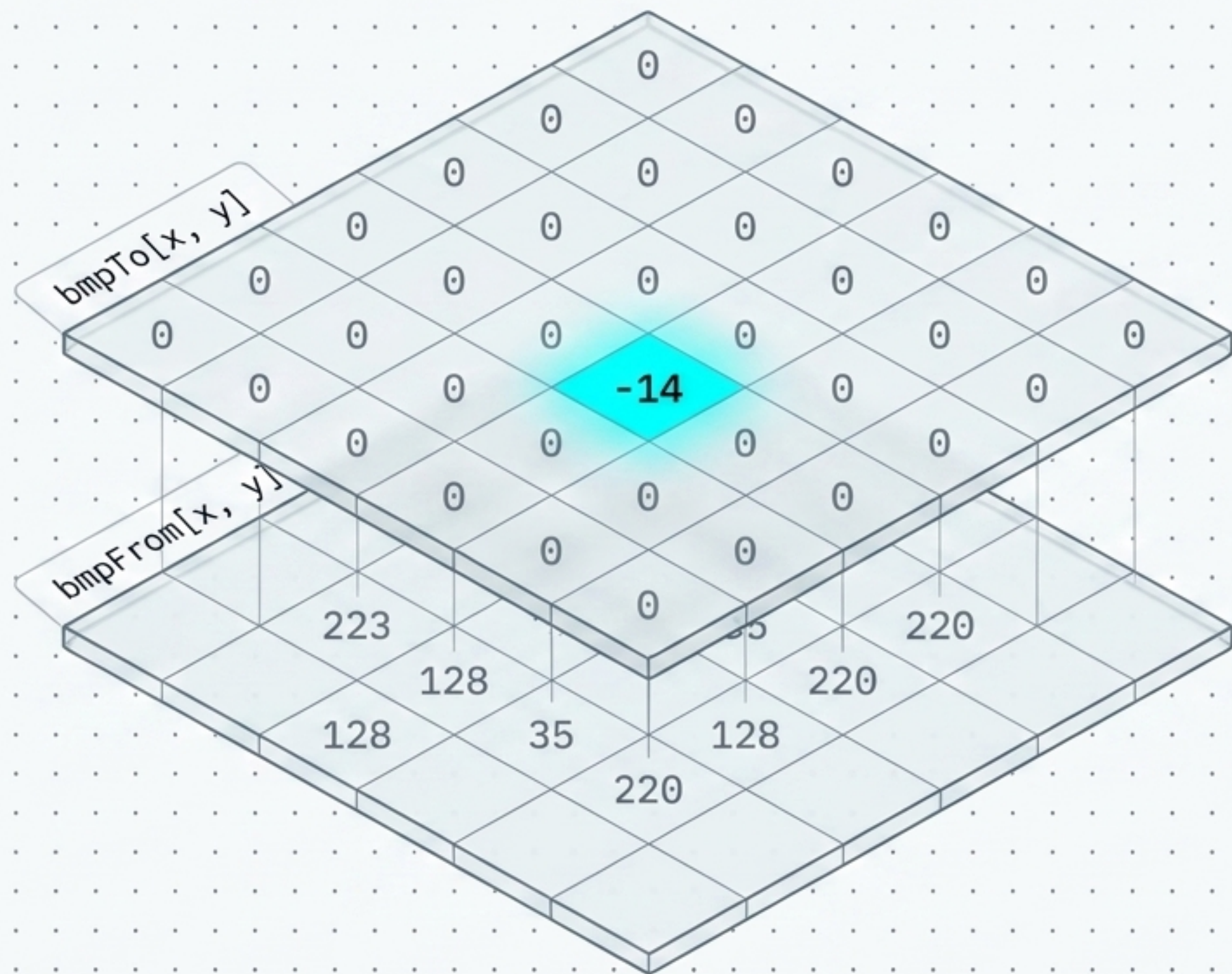
拡散パターン（フロイド-スタインバーグ風）



Matrix Breakdown

1	分子 (ratio):	[7, 3, 1] — 注目ピクセルに近いほど、多くの誤差を配分する。
2	分母 (denominator):	16 — 配分比率の合計。
3	相対座標 (tdx, tdy):	X方向 [1, -1, 0, 1] / Y方向 [0, 1, 1, 1]

データアーキテクチャ：2つの配列



`bmpFrom`

変換前画像の輝度配列（固定データ / 0~255）

`bmpTo`

変換後輝度配列。初期値はすべて0だが、処理を進めるにつれて周囲から受け取った「誤差」が蓄積されていく動的なバッファ。

全体アーキテクチャ（メインループ）

4つの論理ブロックによるループ構造



Block 01: 輝度の統合 (F値計算)

単純に元の輝度を見るだけではない。過去に処理した周囲のピクセルから押し付けられた「誤差の合計 (bmpTo)」を、元の輝度 (bmpFrom) に合算することで、現在の真の輝度 f を導き出す。

```
for y in range(height):  
    for x in range(width):  
        # Block 01: 輝度の統合  
        f = bmpFrom[x, y] + bmpTo[x, y]
```

$$\begin{array}{c} \text{bmpFrom} \\ \boxed{128} \end{array} + \begin{array}{c} \text{bmpTo} \\ \boxed{-14} \end{array} = \begin{array}{c} f \\ \boxed{114} \end{array}$$

Block 02: 2値化と誤差算出

中間値である 128 を閾値として白黒を判定。bmpTo の該当座標を0か255で書きする。出力値と元の f の差分が、新たに発生した誤差 d となる。

```
# Block 02: 2値化と誤差算出
```

```
if f >= 128:
```

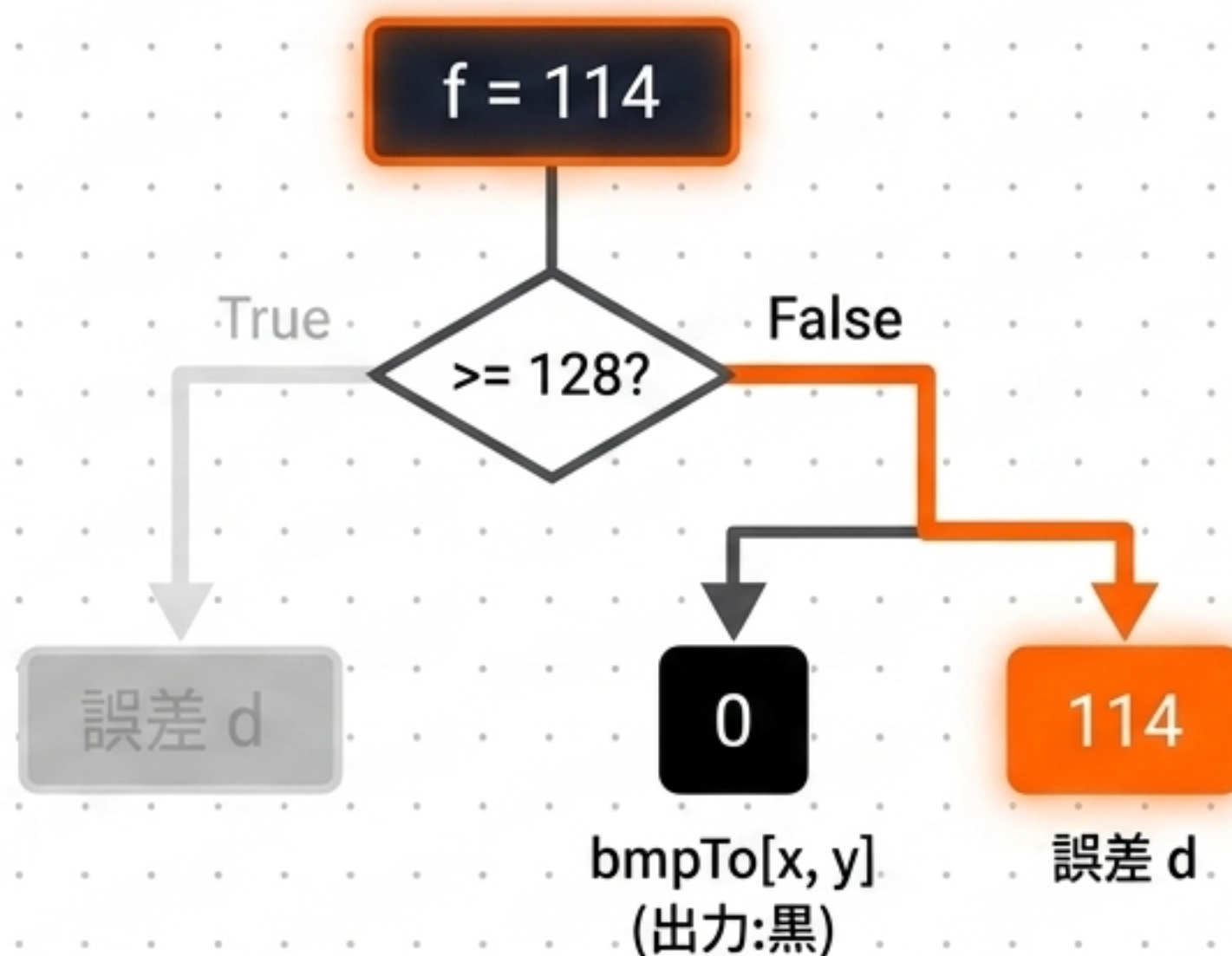
```
    d = f - 255
```

```
    bmpTo[x, y] = 255
```

```
else:
```

```
    d = f
```

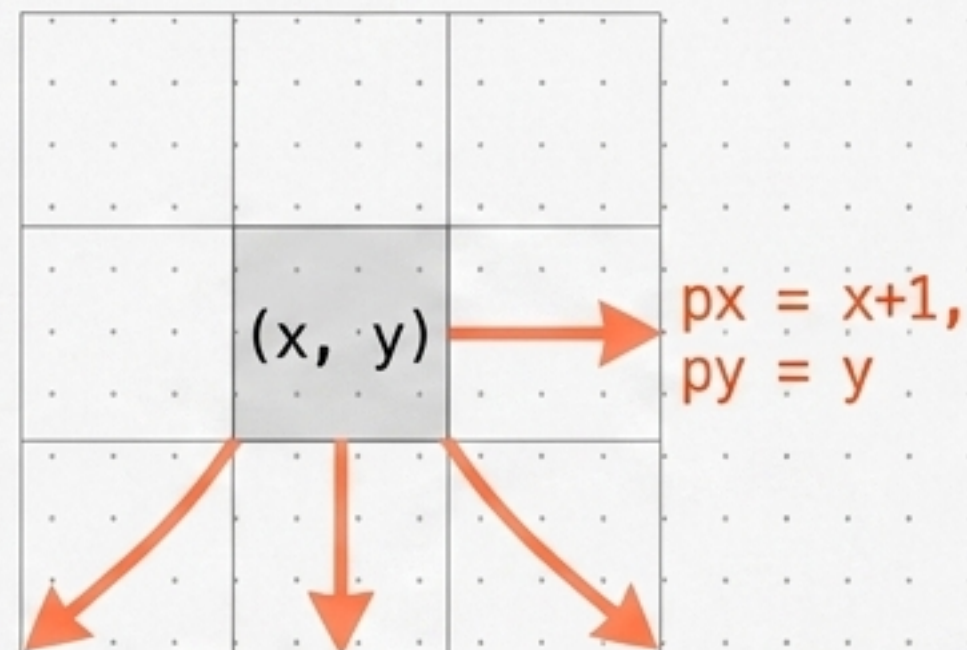
```
    bmpTo[x, y] = 0
```



Block 03: 拡散先ループ

発生した誤差 d を周囲4箇所に分けるための内部ループ。相対座標配列 tdx, tdy を用いて、拡散対象となるピクセルの絶対座標 px, py を算出する。

```
# Block 03: 拡散先座標の計算
for c in range(ratioCount): # c = 0, 1, 2, 3
    px = x + tdx[c]
    py = y + tdy[c]
```

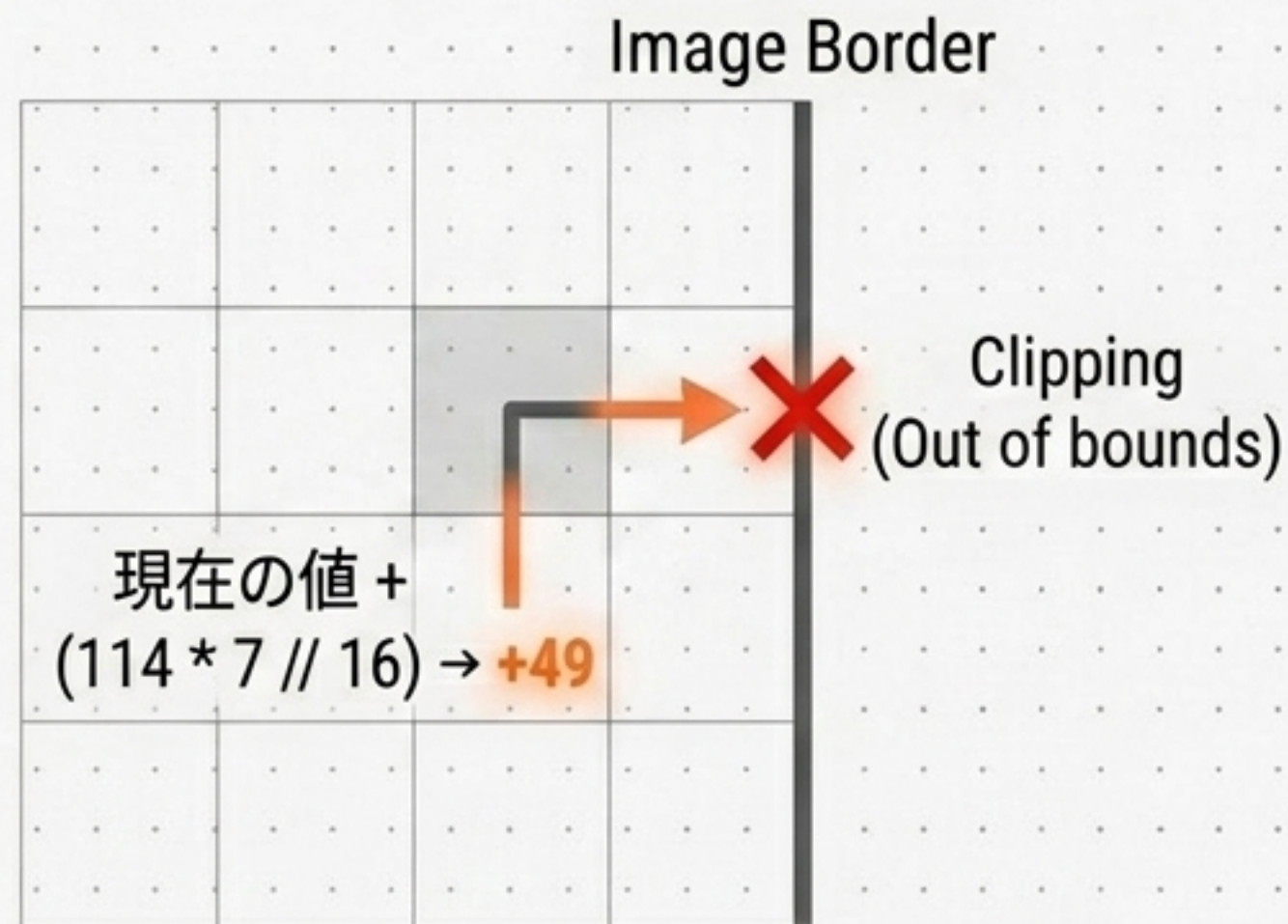


$px = x-1, py = y+1$ $px = x, py = y+1$ $px = x+1, py = y+1$

Block 04: クリッピングと誤差加算

画像の境界外に対する処理を防ぐ（クリッピング判定）。範囲内であれば、事前に定義された配分比率 $\text{ratio}[c] / 16$ に従って計算した誤差を、`bmpTo` の該当座標に「加算（+=）」して蓄積させる。

```
# Block 04: クリッピングと誤差加算
if (0 <= px < width) and (0 <= py < height):
    # 誤差を配分してバッファに加算
    bmpTo[px, py] += d * ratio[c] // 16
```



実行トレース（左上から3ピクセル目の処理例）

理論が実際にどう動くか。注目ピクセルの元の値は128だが、左から来たマイナス誤差(-14)により114に低下し、黒と判定される。その結果生じた特大のプラス誤差(114)が、次のピクセルへと伝播していく。

変換前画像 (bmpFrom)

0	128	0	128	124
128	128	114	124	113
115	126	114	130	128
118	128	114	139	114
119	124	128	114	114

Step 1:
 $F = 128$ (bmpFrom)
 $+ (-14)$ (bmpTo)
 $= 114$

0	128	0	128	124
128	128	114	124	113
115	126	Threshold 128	130	128
118	128	114	139	114
119	124	128	114	114

Step 2:
 $114 < 128$
-> 出力 0 (Black), $D = 114$

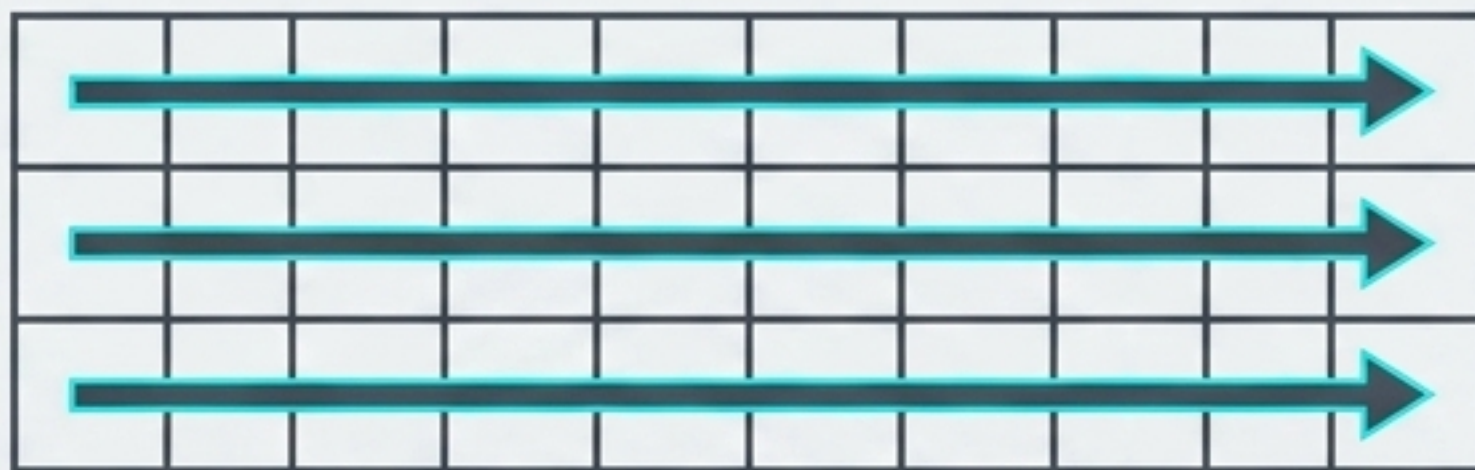
変換後輝度配列 (bmpTo)

0	128	0	128	124
0	0	+49	+49	116
0	128	114	+16	128
0	128	114	+49	114
0	100	123	+19	128

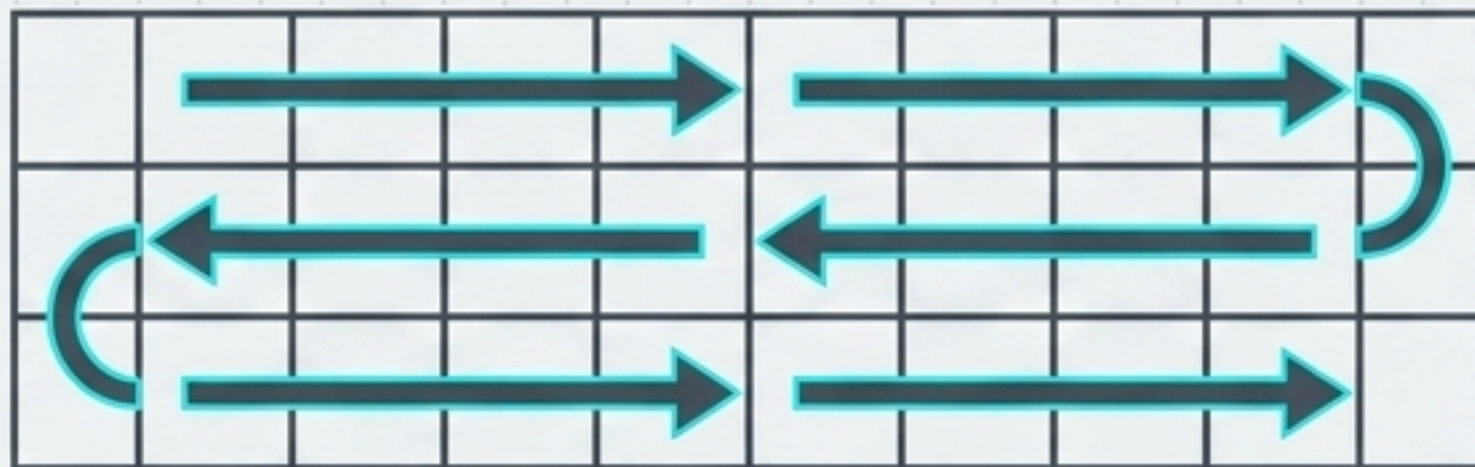
Step 3: 右への拡散
 $D * 7/16 = 114 * 7 / 16 = 49$

Advanced: Serpentine Scanning (画質向上のための改修)

常に左から右へ処理すると、誤差が特定方向に偏り特有の模様（アーティファクト）が生じる。Y座標が偶数の行では処理を右から左へ逆転（ $y \bmod 2$ ）させることで、誤差の拡散方向を散らし、より自然なノイズを生成する。

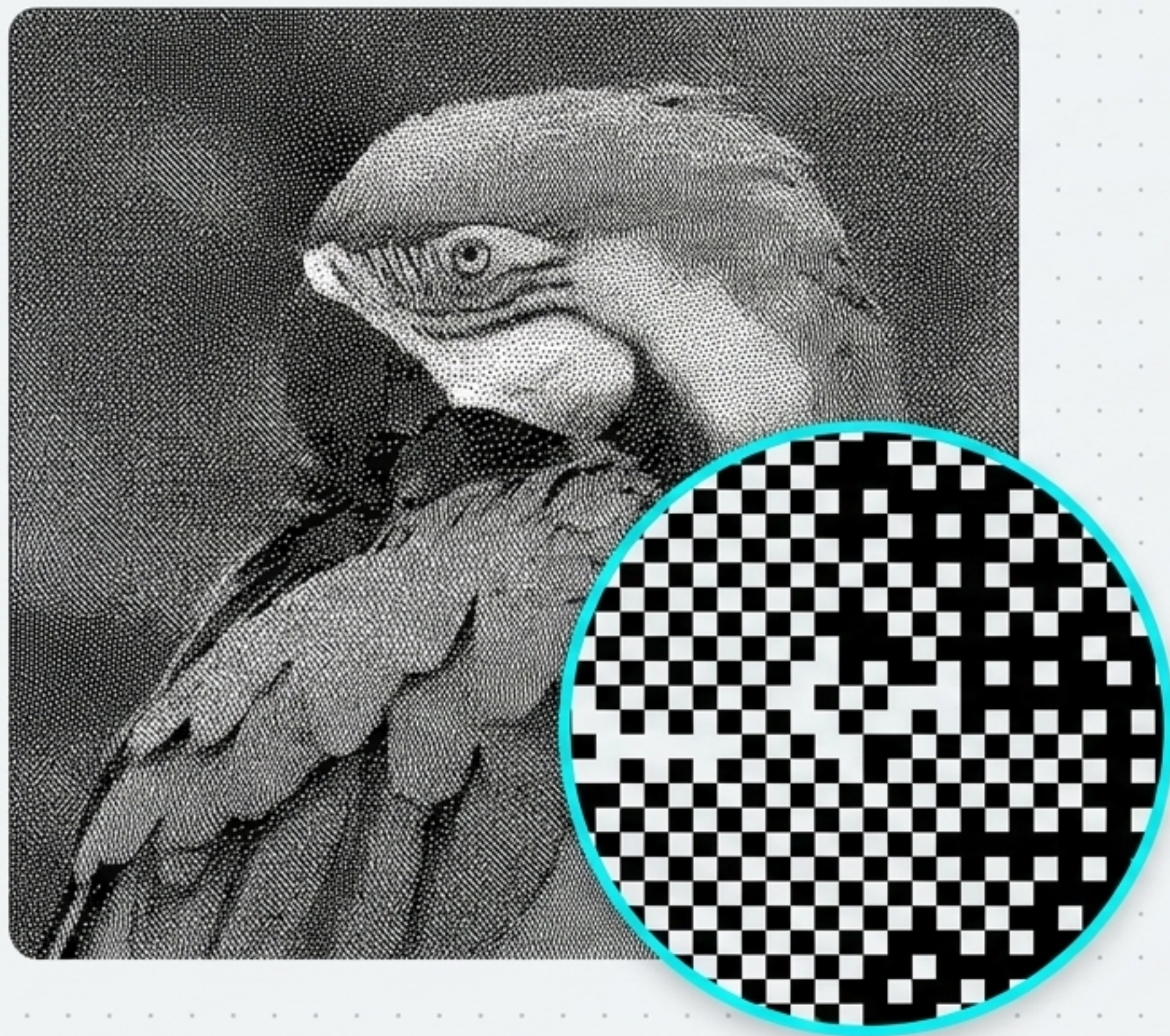


Top Grid (Standard Scan): 誤差の偏りによるアーティファクトが発生しやすい



Bottom Grid (Serpentine Scan): Y座標の偶奇($y \bmod 2$)で方向を反転し、画質を向上

局所的なルールが、大域的な美しさを生む



状態の分離

bmpFrom（静的）と bmpTo（動的）を分けることで状態管理をシンプル化。

エネルギー保存

閾値で削られた輝度情報を「誤差」として次へ渡すことで、画像全体の明るさを厳密に保存。

アルゴリズムの力

単純な四則演算とif文の繰り返しが、ハードウェアの限界 (2値表示) を超えた視覚表現を可能にする。