Halftoning

Halftoning é o processo usado para converter uma imagem em níveis de cinzas *G* numa imagem binária *B* de forma que *B* pareça *G* se *B* for vista a uma certa distância. A imagem *B* é denominada de imagem halftone ou imagem meio-tom. Halftoning é utilizado, por exemplo, para imprimir imagens em níveis de cinzas em impressoras a laser ou jato de tinta, que só conseguem imprimir pontos pretos minúsculos.

Halftoning colorido é usado para converter uma imagem colorida em diversas imagens binárias. Por exemplo, uma imagem colorida pode ser convertida em 4 imagens binárias CMYK (cyan, magenta, yellow and black) cada uma representando uma cor da impressora jato de tinta.

Alguns algoritmos de halftoning (por exemplo, difusão de erro) também podem ser utilizados para minimizar a distorção resultante de quantização, quando a palete utilizada contiver poucas cores. Veja a transparência sobre “escolha de palete e quantização”.

Formalização do problema de halftoning:

Dada uma imagem em níveis de cinza *G* com valores reais entre 0 e 1, construir uma imagem binária *B* com valores 0 ou 1 tal que

$$\bar{B}(l,c) \approx G(l,c)$$

onde $\bar{B}(l,c)$ é a média dos valores de *B* em torno do pixel (*l*,*c*).

Thresholding (limiarização)

A primeira idéia para fazer halftoning é simplesmente limiarizar a imagem *G*. Isto é, um pixel de saída *B*(*l*,*c*) recebe a cor preta se *G*(*l*,*c*) estiver abaixo de um limiar (e vice-versa).

LENNA.TGA original

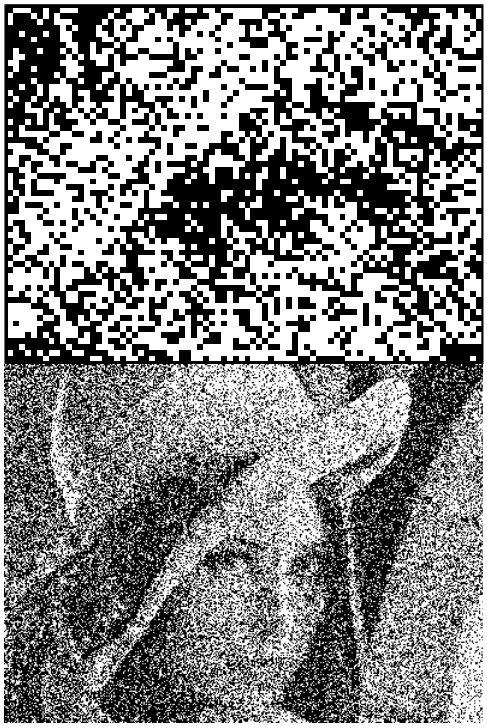
Thresholding (limiarização simples)

A limiarização simples gera regiões pretas e brancas. Não é possível observar as tonalidades de cinza.

Limiarização com ruído

Para que as diferentes tonalidades de cinza sejam visíveis, é possível acrescentar um ruído na imagem antes de fazer a limiarização.

Thresholding (limiarização simples)



Limiarização após acrescentar ruído uniforme.

gauss.tga

gauss.bmp (gauss.tga binarizado)

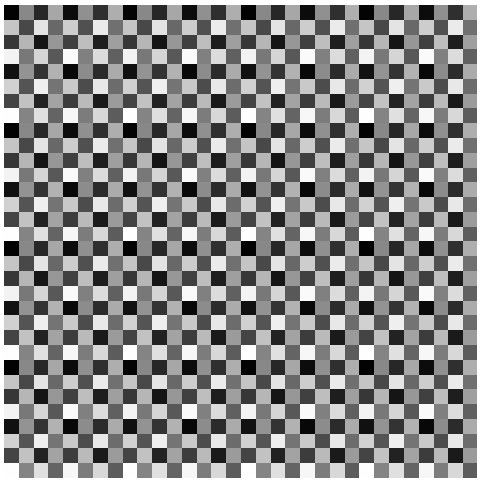
Com essa técnica, é possível observar diferentes níveis de cinza. Porém, a imagem obtida é ruidosa, pois foi acrescentado ruído.

Ordered dithering, dispersed dots

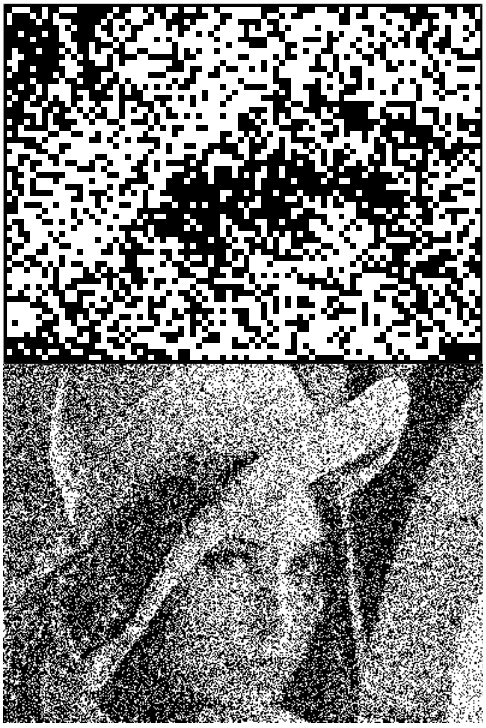
Para que a imagem halftone não se torne ruidosa, é possível acrescentar ruído de forma ordenada. Este processo chama-se “ordered dithering”.

```
//Ordered dithering - pontos dispersos - pos-2015 - testado 2016
#include <cekeikon.h>
int main(int argc, char** argv)
{ if (argc!=3) erro("Dispersos ent.pgm sai.pbm");
  Mat_<GRY> ent; le(ent,argv[1]);
  Mat_<GRY> sai(ent.rows,ent.cols);
  ImgLcr<GRY> D( 8, 8, {
     0, 32,  8, 40,  2, 34, 10, 42,
    48, 16, 56, 24, 50, 18, 58, 26,
    12, 44,  4, 36, 14, 46,  6, 38,
    60, 28, 52, 20, 62, 30, 54, 22,
     3, 35, 11, 43,  1, 33,  9, 41,
    51, 19, 59, 27, 49, 17, 57, 25,
    15, 47,  7, 39, 13, 45,  5, 37,
    63, 31, 55, 23, 61, 29, 53, 21});
  for (int l=0; l<ent.rows; l++)
    for (int c=0; c<ent.cols; c++) {
      if (ent(l,c)+(4*D(l,c)+2-128) < 128)
        sai(l,c)=0;
      else sai(l,c)=255;
    }
  imp(sai,argv[2]);
}
```

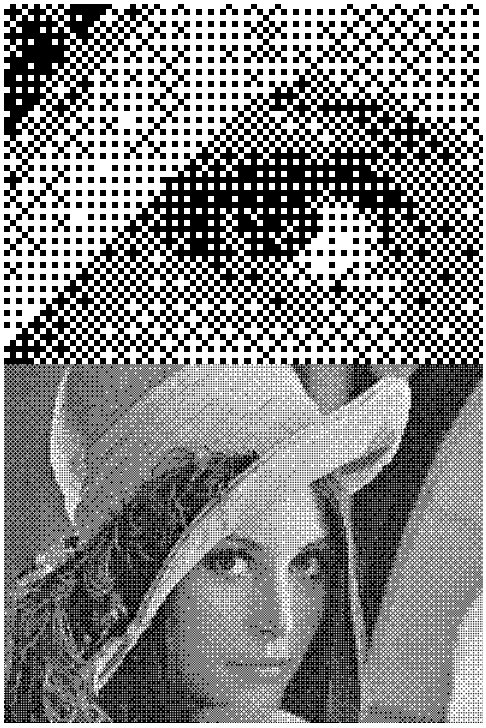
- Nota: Não é necessário calcular explicitamente os índices da imagem D, pois estamos acessando a imagem D no modo ‘atr’ (replicado).
- Nota: A matriz de ruído ordenado D foi projetada para gerar imagem halftone com pontos dispersos.
- Nota: O ruído inserido é (4*D(l2,c2)+2-128). O ruído pode ir de -126 até +126.
- Nota: Alguns “drivers” de impressoras jatos de tinta usam o método “ordered dithering, dispersed dots” para imprimir imagens em níveis de cinza.



A matriz de limiar “dispersed dots”, replicada 4×4 vezes, e visualizada como uma imagem.



Ruído branco ±0,5



Ordered dithering dispersed dots

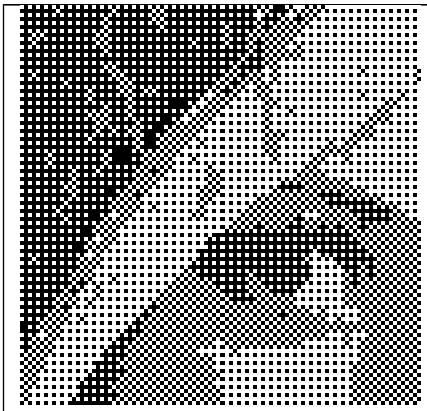
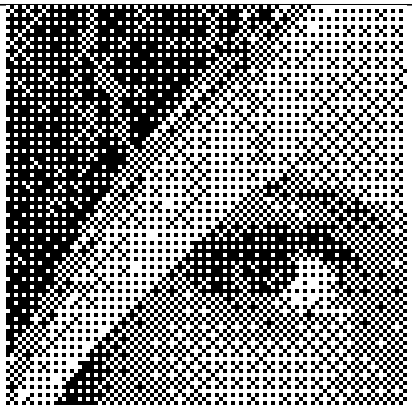
A matriz D 8×8 pode ser obtida de forma recursiva:

Matriz 2×2:

```
IMGGRY D(2,2,
                0,2,
                3,1);
```

Matriz 4×4:

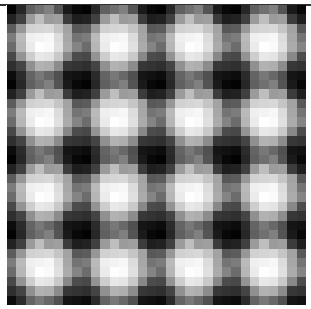
```
IMGGRY D(4,4,
                 0, 8, 2,10,
                12, 4,14, 6,
                 3,11, 1, 9,
                15, 7,13, 5);
```

		
matriz 2×2	matriz 4×4	matriz 8×8

Ordered dithering, clustered dots

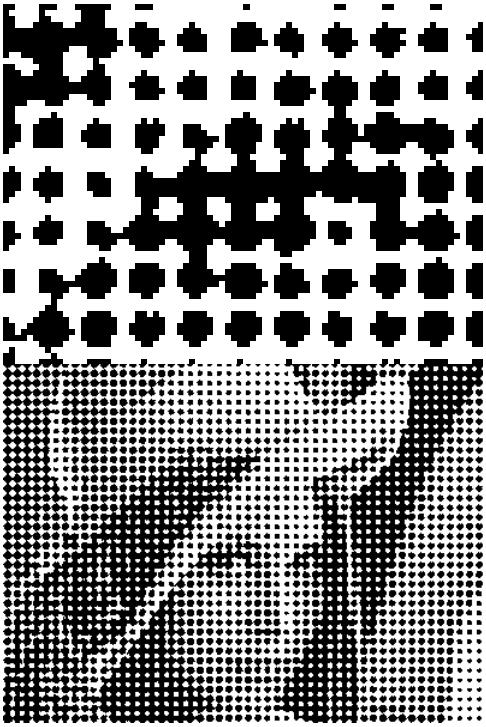
O algoritmo de ordered dithering pode gerar imagem halftone com pontos aglutinados simplesmente mudando a matriz de limiar (ou matriz de ruído) D. Uma possível matriz é:

```
//Clustered dots
  ImgLcr<GRY> D( 8, 8, {
     0,  8, 22, 26, 30, 19,  5,  1,
     7, 14, 37, 46, 37, 38, 13,  6,
    21, 36, 51, 52, 53, 48, 39, 20,
    29, 45, 59, 60, 61, 54, 40, 27,
    25, 44, 58, 63, 62, 55, 41, 31,
    16, 35, 50, 57, 56, 49, 32, 17,
    10, 15, 34, 43, 42, 33, 12, 11,
     2,  9, 23, 28, 24, 18,  4,  3});
```

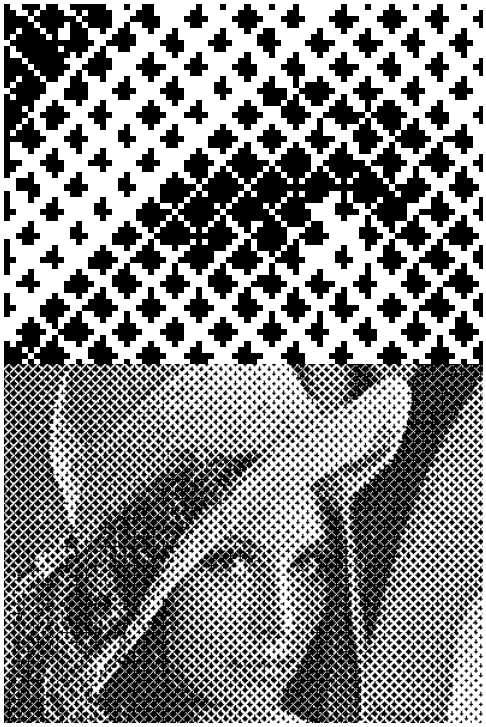


A matriz de limiar “clustered dots”, replicada 4×4 vezes, e visualizada como uma imagem.

Usando a matriz acima, obtém a imagem da coluna da esquerda da figura abaixo. A coluna da direita foi gerada pelo programa “Alchemy”.



Ordered dithering clustered dots
(meu programa)



Ordered dithering clustered dots
(alchemy)

Imagem com pixels em ponto flutuante (Mat_<FLT>)

Certos algoritmos de Processamento de Imagens ficam mais simples, se utilizar imagens em ponto flutuante. Isto evita acumulação de erros de arredondamento.

Cada pixel de Mat_<FLT> é do tipo float (isto é, número em ponto flutuante com 4 bytes), no intervalo [0.0, 1.0]. O preto é 0.0 e o branco é 1.0.

Convertendo uma Mat_<GRY> [0...255] para Mat_<FLT> [0.0, 1.0], um pixel com nível de cinza *n* é mapeado no valor $(n)/255$, isto é, 0 é convertido em 0.0 e 255 é convertido em 1.0.

Nota: A biblioteca Cekeikon consegue ler e escrever imagens em formato de arquivos texto. Basta usar as extensões .TXT ou .MAT (ambos são sinônimos para a biblioteca).

Nota: A biblioteca Cekeikon consegue ler e escrever imagens em formato de arquivos binário. Basta usar a extensão .IMG.

Difusão de erro

A difusão de erro é um outro algoritmo de halftoning. A implementação deste algoritmo fica menos suscetível a erros de arredondamento se utilizar Mat_<FLT>.

A imagem em níveis de cinza *G* é lida e convertida em imagem em ponto flutuante *F*. A imagem *F* é escaneada numa certa ordem, por exemplo, na ordem raster. Cada pixel é aproximado para 0 ou 1, gerando a imagem binária de saída *B*. O erro gerado na aproximação é espalhado para pixels vizinhos ainda não processados, segundo os pesos especificados na matriz de difusão de erro.

Diferentes matrizes de iufusão de erro resultam em diferentes métodos de halftoning. Algumas matrizes de difusão de erro:

$\left[\frac{1}{3}\right]\times\begin{bmatrix}\bullet & 1\\ 1 & 1\end{bmatrix}$	$\left[\frac{1}{2}\right]\times\begin{bmatrix}\bullet & 1\\ 1 & 0\end{bmatrix}$	$\left[\frac{1}{8}\right]\times\begin{bmatrix}\bullet & 3\\ 3 & 2\end{bmatrix}$	$\left[\frac{1}{16}\right]\times\begin{bmatrix}\bullet & 7\\ 3 & 5 & 1\end{bmatrix}$
PSI2651	PSI5796	Rogers	Floyd and Steinberg

$$\left[\frac{1}{48}\right]\times\begin{bmatrix} & & \bullet & 7 & 5\\ 3 & 5 & 7 & 5 & 3\\ 1 & 3 & 5 & 3 & 1\end{bmatrix}$$

Jarvis, Judice and Ninke

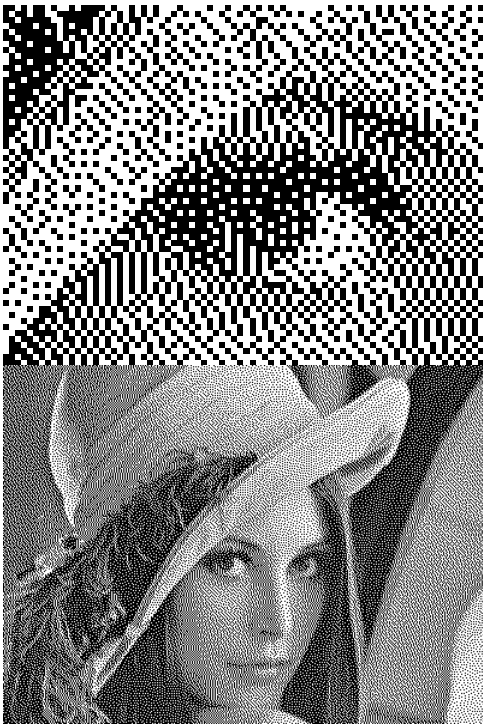
$$\left[\frac{1}{42}\right]\times\begin{bmatrix} & & \bullet & 8 & 4\\ 2 & 4 & 8 & 4 & 2\\ 1 & 2 & 4 & 2 & 1\end{bmatrix}$$

Stucki

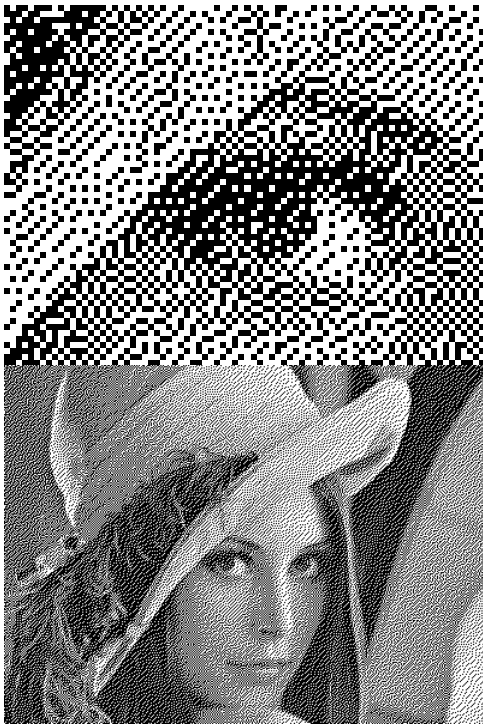
Abaixo, os programas de difusão de erro. Note que não é necessário se preocupar com as bordas da imagem, pois a biblioteca Cekeikon possui `backg`, já discutido.

```cpp
//diferro.cpp - grad2016
#include <cekeikon.h>
int main()
{ ImgAtb<FLT> a; le(a,"lennag.jpg");
  Mat_<FLT> b(a.rows,a.cols);

  for (int l=0; l<a.rows; l++)
    for (int c=0; c<a.cols; c++) {
      if (a(l,c)>0.5) b(l,c)=1;
      else b(l,c)=0;
      float err=(b(l,c)-a(l,c))/3;
      //a(l+1,c-1)=a(l+1,c-1)-err;
      a(l+1,c)=a(l+1,c)-err;
      a(l+1,c+1)=a(l+1,c+1)-err;
      a(l,c+1)=a(l,c+1)-err;
    }
  imp(b,"diferro.pgm");
}
```

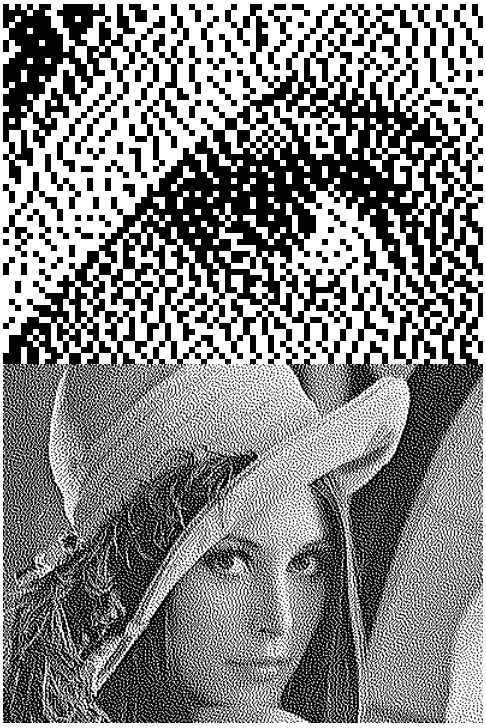
```cpp
//diferr.cpp - pos2016
#include <cekeikon.h>

int main() {
  Mat_<FLT> a; le(a,"lenna.jpg"); //0=preto 1=branco
  Mat_<FLT> b(a.rows,a.cols,1.0);
  for (int l=0; l<a.rows-1; l++)
    for (int c=0; c<a.cols-1; c++) {
      if (a(l,c)<0.5) b(l,c)=0.0;
      else b(l,c)=1.0;
      float err=(b(l,c)-a(l,c))/3;
      a(l,c+1)  -= err; // a(l,c+1) = a(l,c+1) - err;
      a(l+1,c)  -= err;
      a(l+1,c+1)-= err;
    }
  imp(b,"diferr.pgm");
}
```



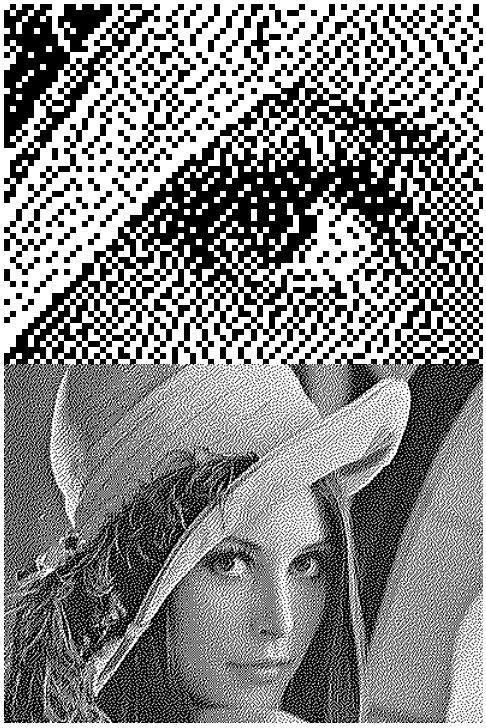
Difusão de erro
(Floyd and Steinberg)



Difusão de erro
(Rogers)



Difusão de erro
(Jarvis, Judice and Ninke)



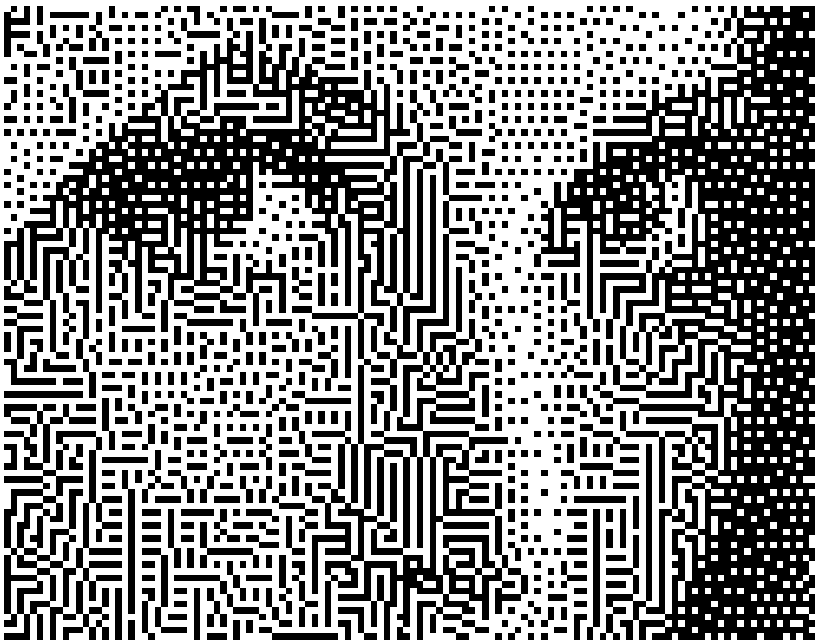
Difusão de erro
(Stucki)

Diferentes matrizes de pesos geram imagens halftone com diferentes texturas. Certos matrizes podem gerar texturas indesejadas, como longas linhas verticais ou horizontais.

Por exemplo, usando a matriz:

$$\left[\frac{1}{4}\right]\times\begin{bmatrix} & \bullet & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

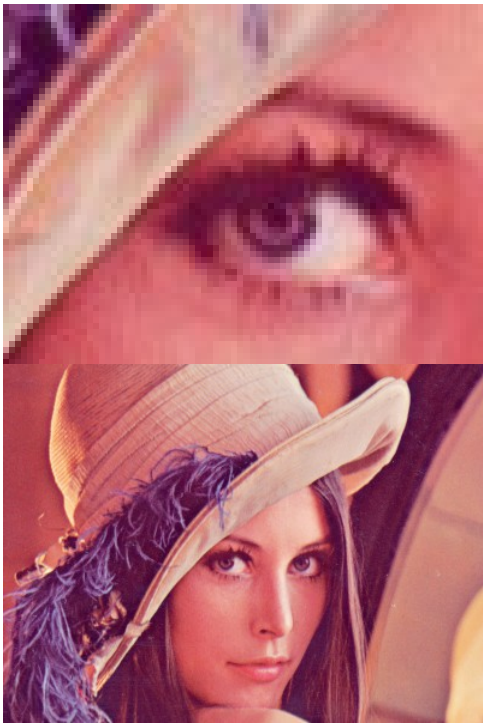
obtemos:



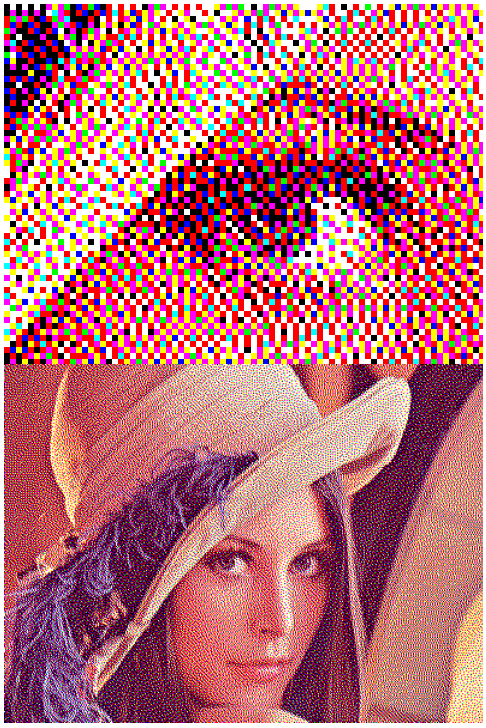
Nota: Faltou comentar sobre o algoritmo de halftoning DBS.
Nota: Watermarking usando diferentes matrizes de difusão de erro.

Difusão de erro de imagens coloridas

Repetindo difusão de erro para as 3 bandas e compondo imagens resultantes, é possível gerar imagens halftone coloridas.



LENNA.TGA original



Difusão de erro
(Floyd and Steinberg)

A difusão de erro também pode ser usada para reduzir o número de bits/pixels, por exemplo, ao converter uma imagem true-color numa imagem com palete. Cada pixel é aproximada pela cor mais semelhante no palete. Depois, o erro cometido ao fazer a quantização é espalhada nos pixels vizinhos nas bandas R, G e B.

Exercício:

Reduzir true color (24 bits/pixel) para 6 bits/pixel usando ordered dithering e difusão de erro.

Dada uma imagem halftone, como descobrir qual foi o método de halftoning usado?

É dada uma imagem halftone *H* que foi gerada pela difusão de erro usando uma entre duas possíveis matrizes de difusão. É possível descobrir qual das duas matrizes foi usada olhando somente *H*? Como isto poderia ser usada para esteganografia, isto é, para esconder um bit numa imagem?

Dada uma imagem quantizada, como descobrir qual foi o palete utilizado? Isto pode ser usado para esteganografia, isto é, para esconder bits numa imagem?