PROCESSAMENTO DE IMAGENS APLICADO A AGROINDUSTRIA

Pedro Luiz de Paula Filho

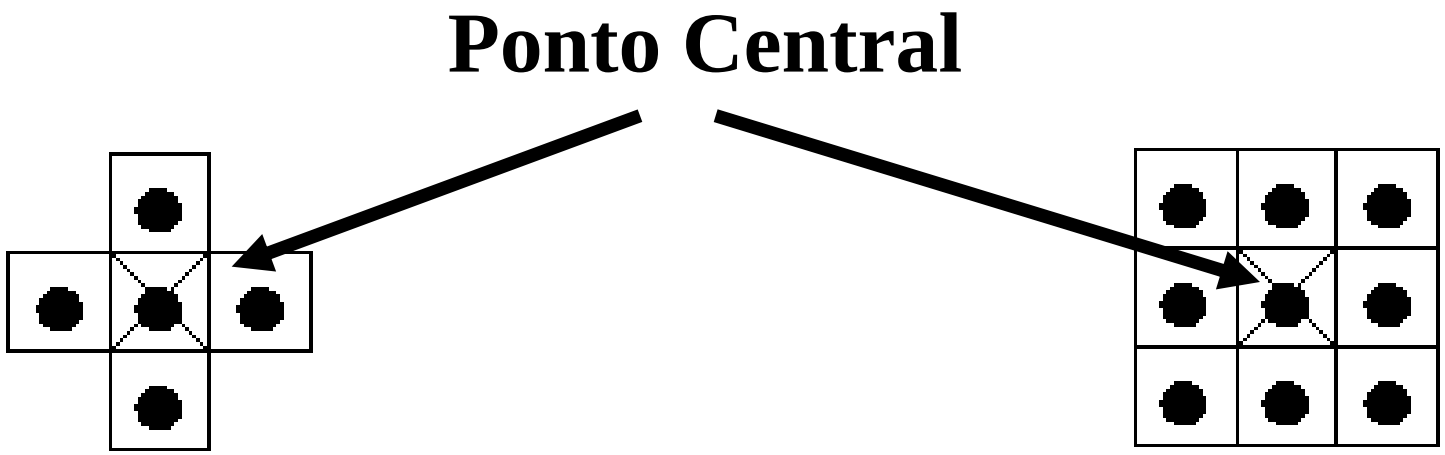
Morfologia Matemática

- Enfoca a estrutura geométrica de uma imagem.
- Faz transformações em uma imagem através de diferentes padrões de formatos conhecidos como elementos estruturantes.
- Pode ser trabalhada em imagens binárias, tons de cinza e coloridas

Elemento Estruturante (EE)

- São formas geométricas simples e menores que a imagem original.
- Conforme o propósito o padrão de EE pode mudar
- O EE é movido sobre a imagem bidimensional e a análise da imagem é baseada no “encaixe” dele dentro da imagem.
- As operações são sempre realizadas em relação à origem (ponto central) do EE.

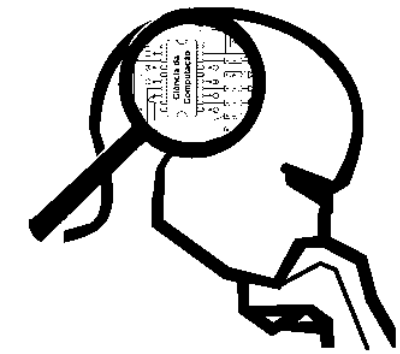
Elemento Estruturante (EE)



Operações Básicas

- Erosão
  - A região mais escura cresce sobre a mais clara
- Dilatação
  - A região mais clara cresce sobre a mais escura

Original



Dilatação

Erosão

Erosão

- Tem como objetivo eliminar dados em uma imagem que não são semelhantes a um tipo padrão.
- Elimina picos positivos mais finos do que o elemento estruturante e expande os picos negativos reduzindo a imagem

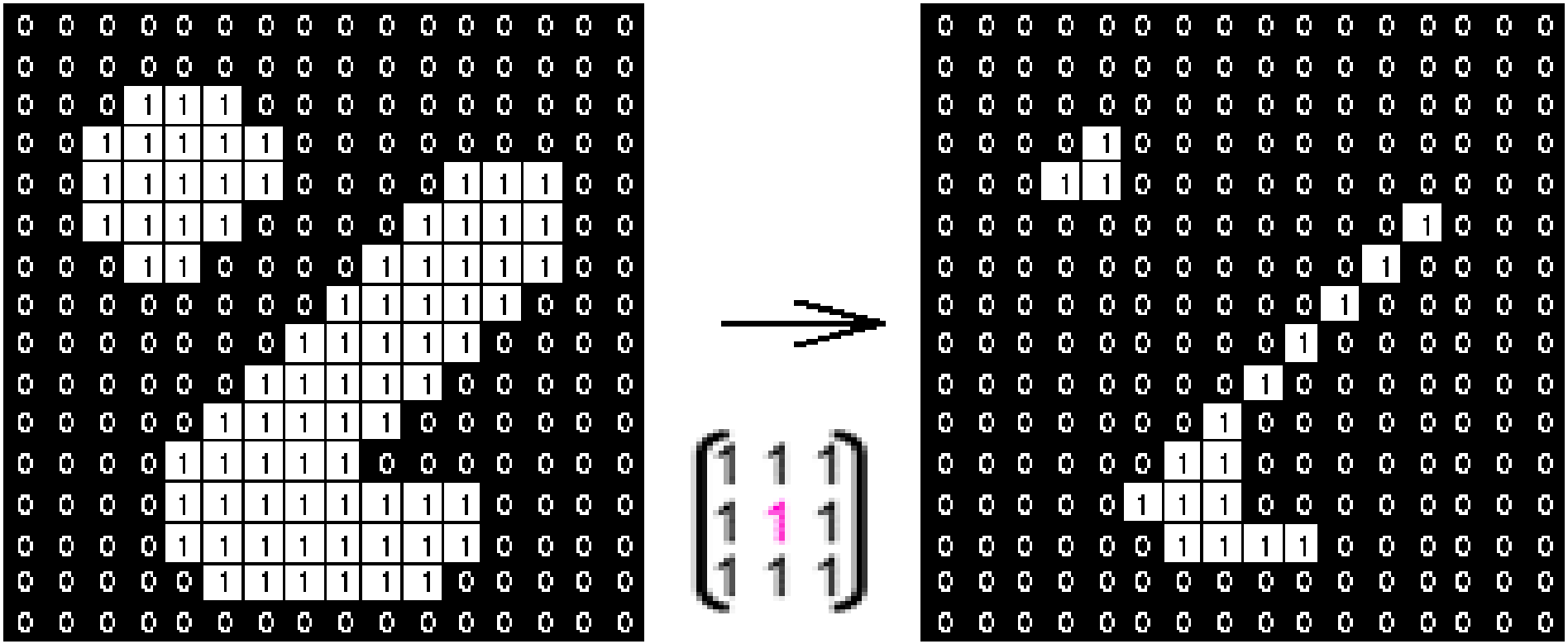
Erosão

- A erosão é definida por:

$$A \ominus B = \{ x : Bx \subset A \}$$

  - ***A*** → Imagem de entrada
  - ***B*** → Elemento estruturante
- A erosão de A por B é o conjunto de todos os pontos x tais que B, quando transladado por x fique contido em A
- Se a origem está no interior do EE a imagem resultante estará no interior da imagem original.

Erosão



Erosão - OpenCv

- Sintaxe:

void erode(InputArray src, OutputArray dst, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue())

  - Kernel → Elemento Estruturante
  - Anchor → posição "central" do EE (Default (-1, -1))
  - Iterations → número de repetições da erosão
  - borderType / borderValue → tratamento borda

Erosão - OpenCv

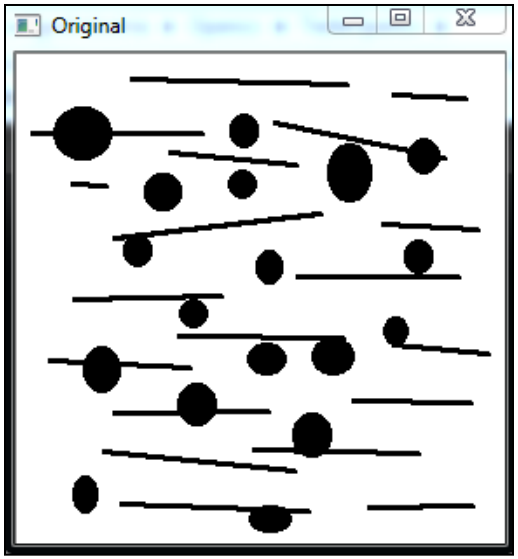
- Kernel → Elemento Estruturante
  - Default 3x3
  - Para criar um elemento estruturante próprio:

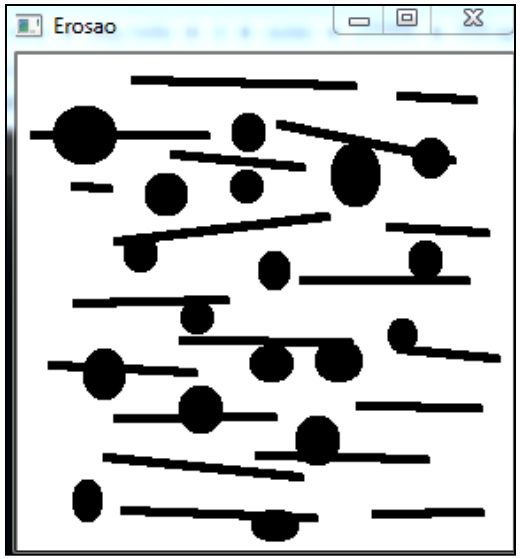
    Mat getStructuringElement(int shape, Size ksize, Point anchor=Point(-1,-1))

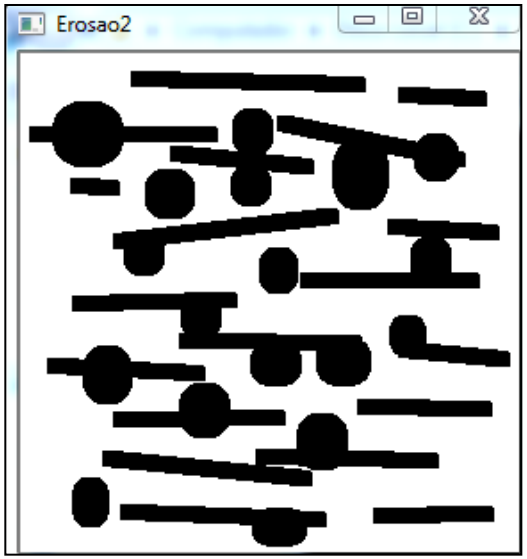
    - Shape → MORPH_RECT, MORPH_ELLIPSE, MORPH_CROSS, CV_SHAPE_CUSTOM
    - Ksize → tamanho do EE
    - Anchor → ponto de ancoragem do EE

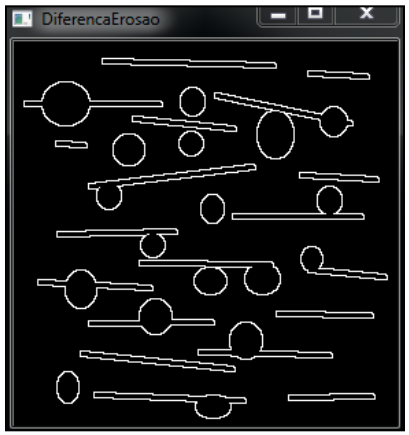
```
Mat elemento = getStructuringElement(MORPH_RECT, Size(3,3), Point(-1,-1));
```

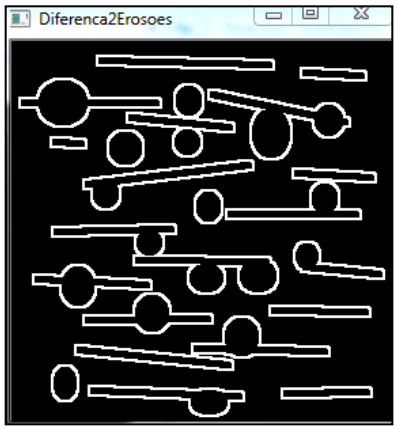
```
Mat image = imread("binbolin.bmp", CV_LOAD_IMAGE_GRAYSCALE);
Mat erosao, erosao2, diferencaErosao, diferenca2Erosoes;
// tipos de EE - MORPH_RECT, MORPH_ELLIPSE, MORPH_CROSS
Mat elemento = getStructuringElement(MORPH_RECT, Size(3,3), Point(-1,-1));
erode(image, erosao , Mat());
erode(image, erosao2, elemento, Point(-1,-1), 3);
diferencaErosao    = image - erosao;
diferenca2Erosoes  = erosao - erosao2;
```











Dilatação

- A dilatação é definida por:

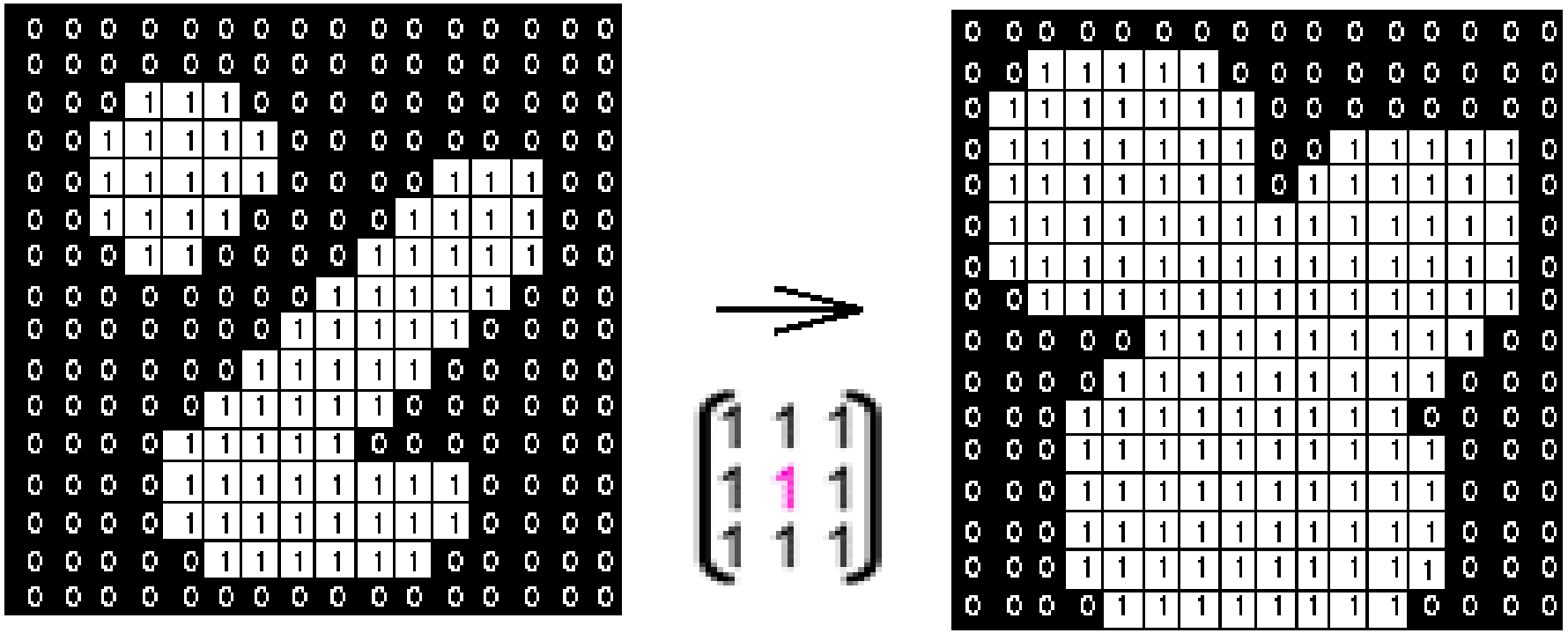
$$A \oplus B = \{ x \in A : Bx \cap x \neq \varnothing \}$$

  - ***A*** → Imagem de entrada
  - ***B*** → Elemento estruturante
- A dilatação de A por B é então o conjunto de todos os deslocamentos x tais que A sobreponham-se em pelo menos um elemento não nulo.
- Diferentemente da erosão, a dilatação é comutativa.

$$A \oplus B = B \oplus A$$

Dilatação

- O resultado da dilatação será uma imagem "engordada"



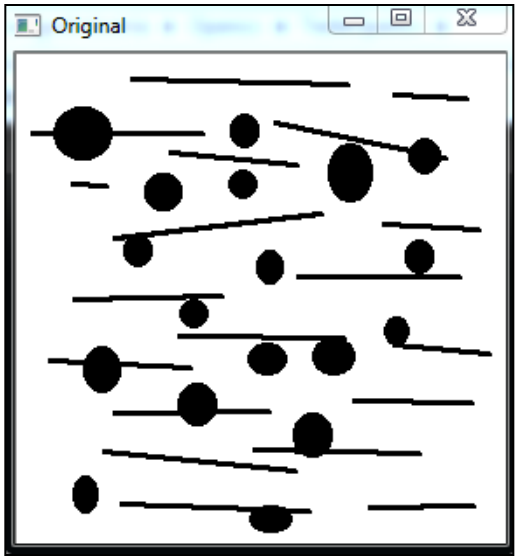
Dilatação - OpenCv

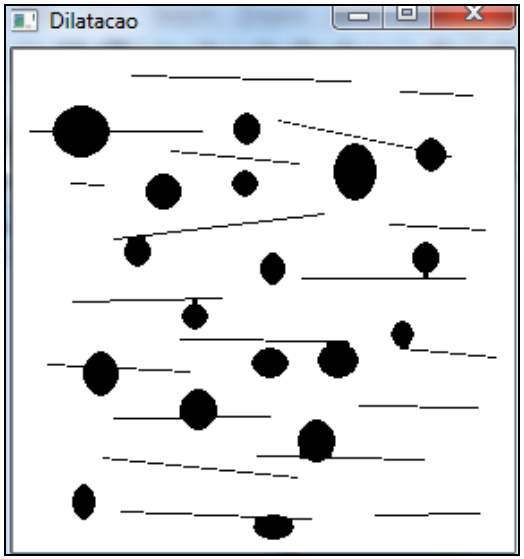
- Sintaxe:

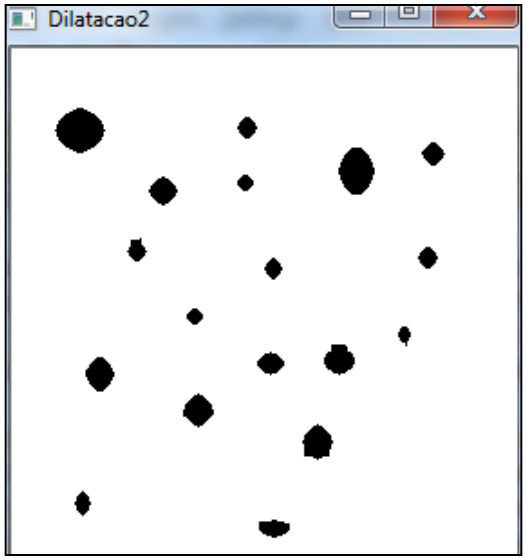
void dilate(InputArray src, OutputArray dst, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue())

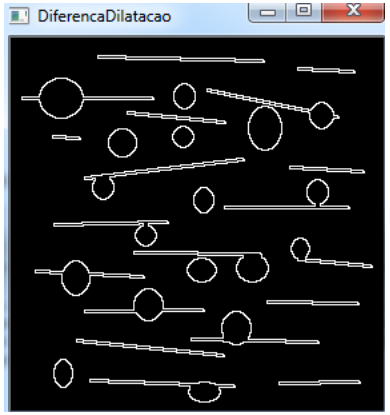
  - Kernel → Elemento Estruturante
  - Anchor → posição "central" do EE (Default (-1, -1))
  - Iterations → número de repetições da erosão
  - borderType / borderValue → tratamento borda

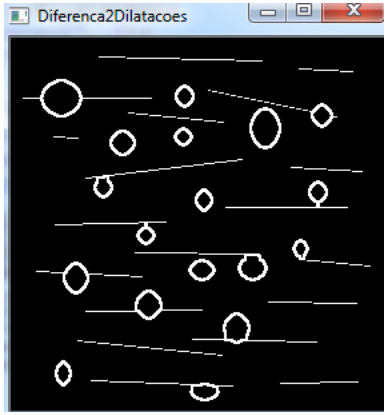
```
Mat image = imread("binbolin.bmp", CV_LOAD_IMAGE_GRAYSCALE);
Mat dilatacao, dilatacao2, diferencaDilatacao, diferenca2Dilatacoes;
// tipos de EE - MORPH_RECT, MORPH_ELLIPSE, MORPH_CROSS
Mat elemento = getStructuringElement(MORPH_RECT, Size(3,3), Point(-1,-1));

dilate(image, dilatacao , Mat());
dilate(image, dilatacao2, elemento, Point(-1,-1), 3);
diferencaDilatacao    = dilatacao - image;
diferenca2Dilatacoes  = dilatacao2 - dilatacao;
```





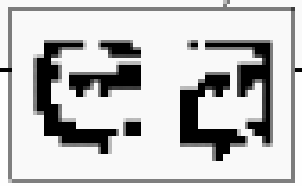






Dilatação

Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

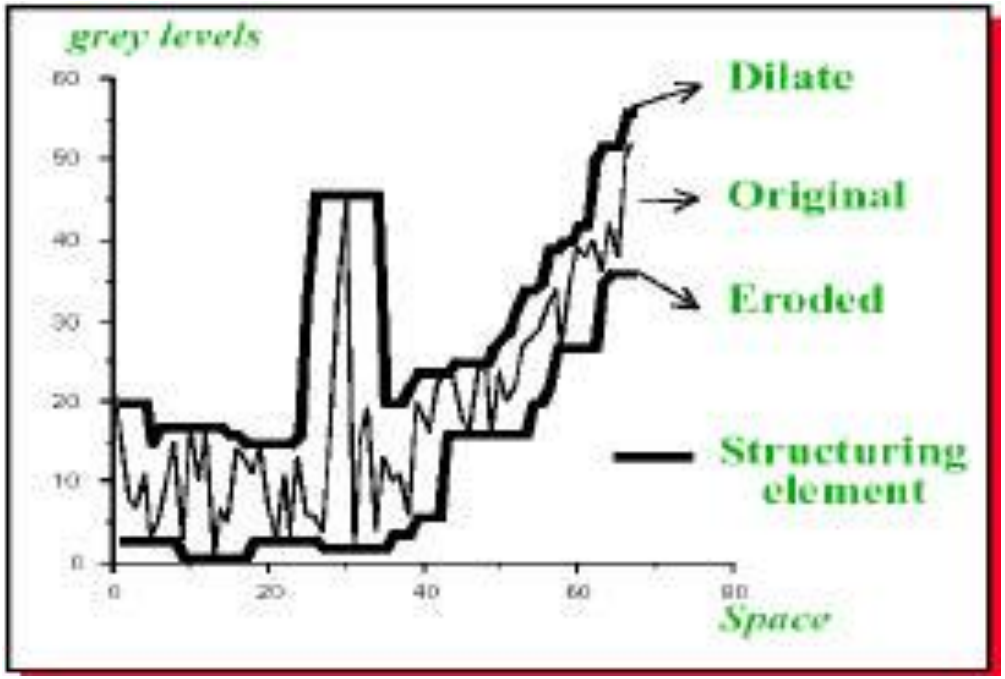


Historically, certain computer programs were written using only two digits rather than four to define the applicable year. Accordingly, the company's software may recognize a date using "00" as 1900 rather than the year 2000.

0	1	0
1	1	1
0	1	0

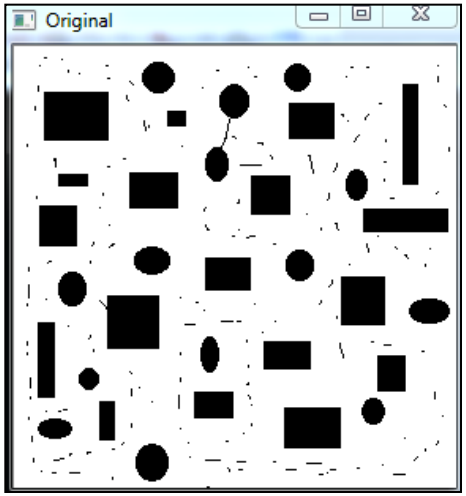
Comparação Erosão-Dilatação

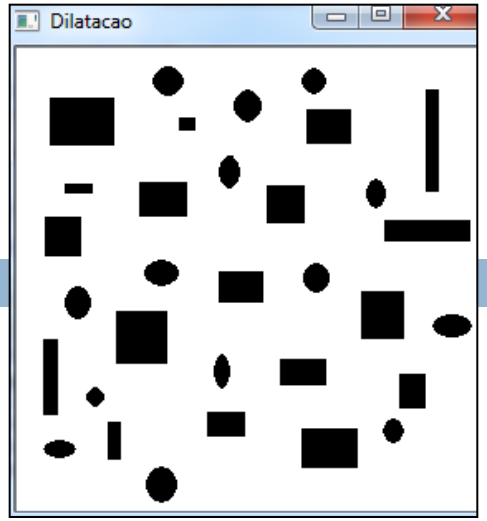
- **Erosão:** elimina picos positivos mais finos do que o EE e expande os picos negativos **Dilatação:** elimina picos negativos mais finos do que o EE e expande os picos positivos

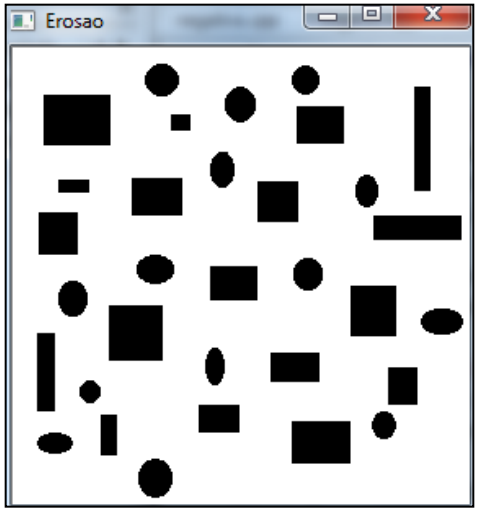


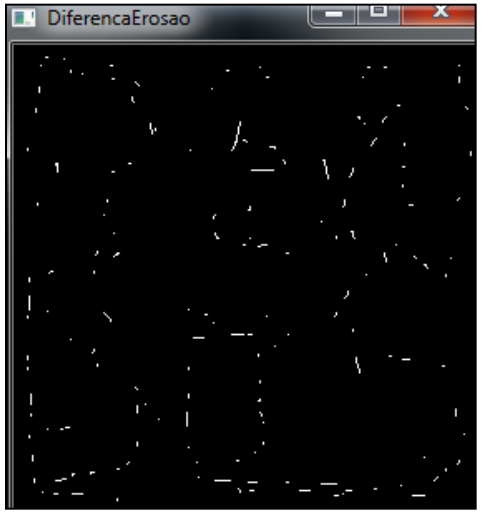
Operações Morfológicas

- A mescla de erosão e dilatação nos traz resultados bastante interessantes, quando o objetivo é extrair informações não desejadas



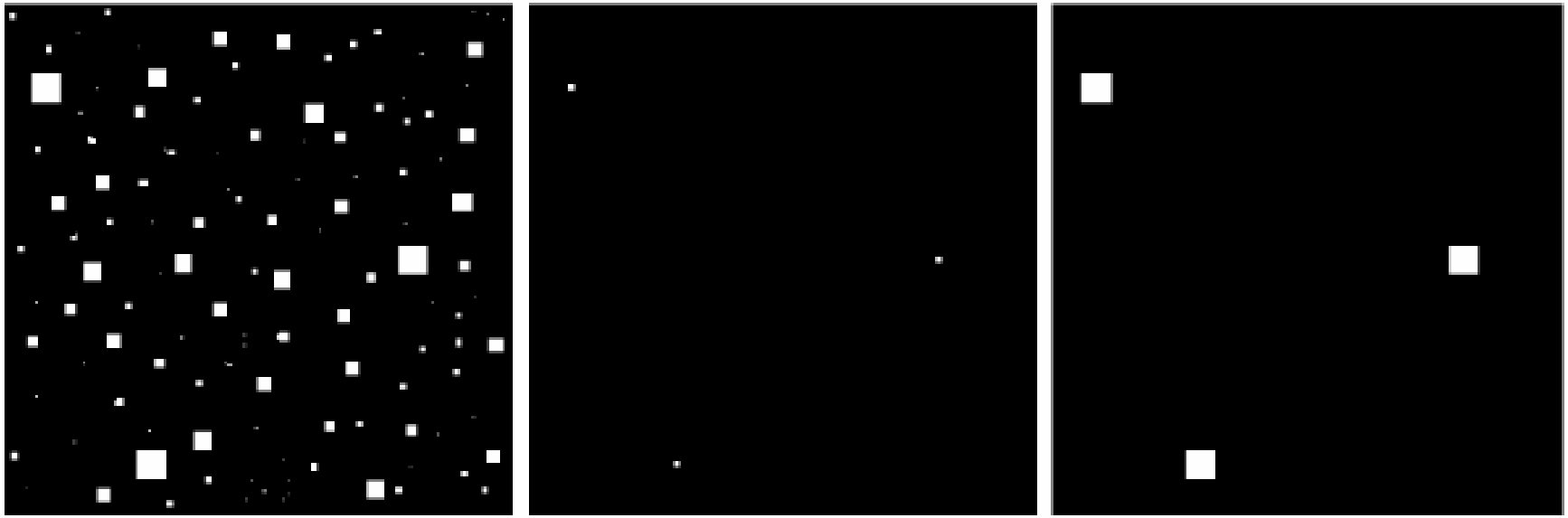







```
Mat image = imread("BINQUAD3.bmp", CV_LOAD_IMAGE_GRAYSCALE);
Mat erosao, diferencaErosao;
Mat dilatacao, diferencaDilatacao;
dilate(image, dilatacao , Mat());
diferencaDilatacao    = dilatacao - image;
erode(dilatacao, erosao , Mat());
diferencaErosao    = erosao - image;
```

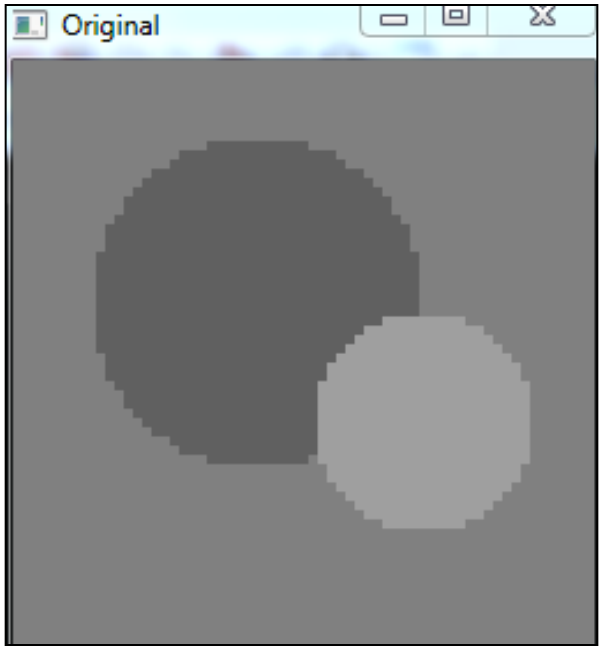
Operações Morfológicas

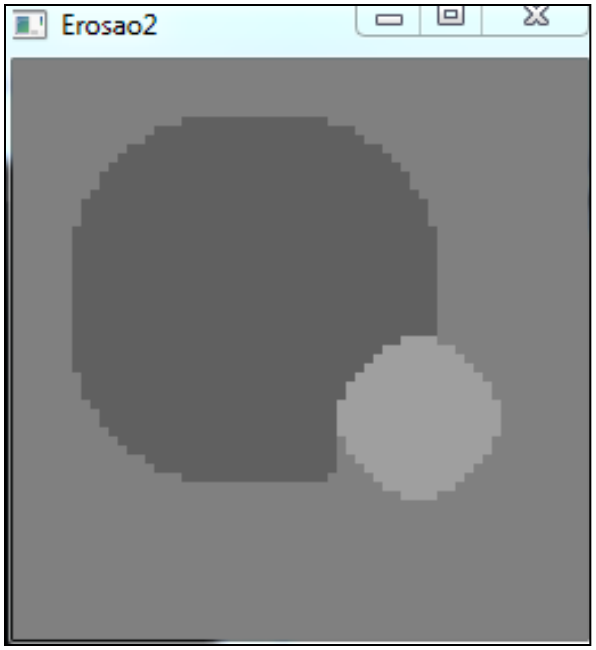


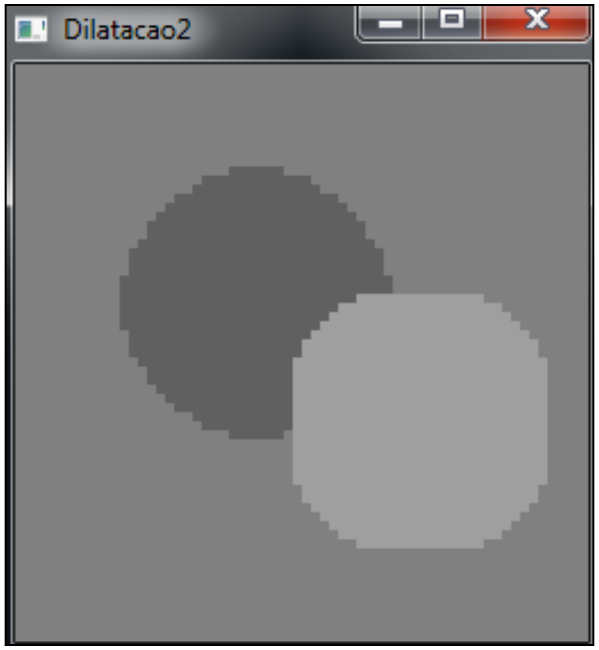
Original Erosão Dilatação

- Elemento estruturante → MORPH_RECT (3x3)
- Aplicado por 13x tanto na erosão quanto na dilatação

Operações Morfológicas - Cinza

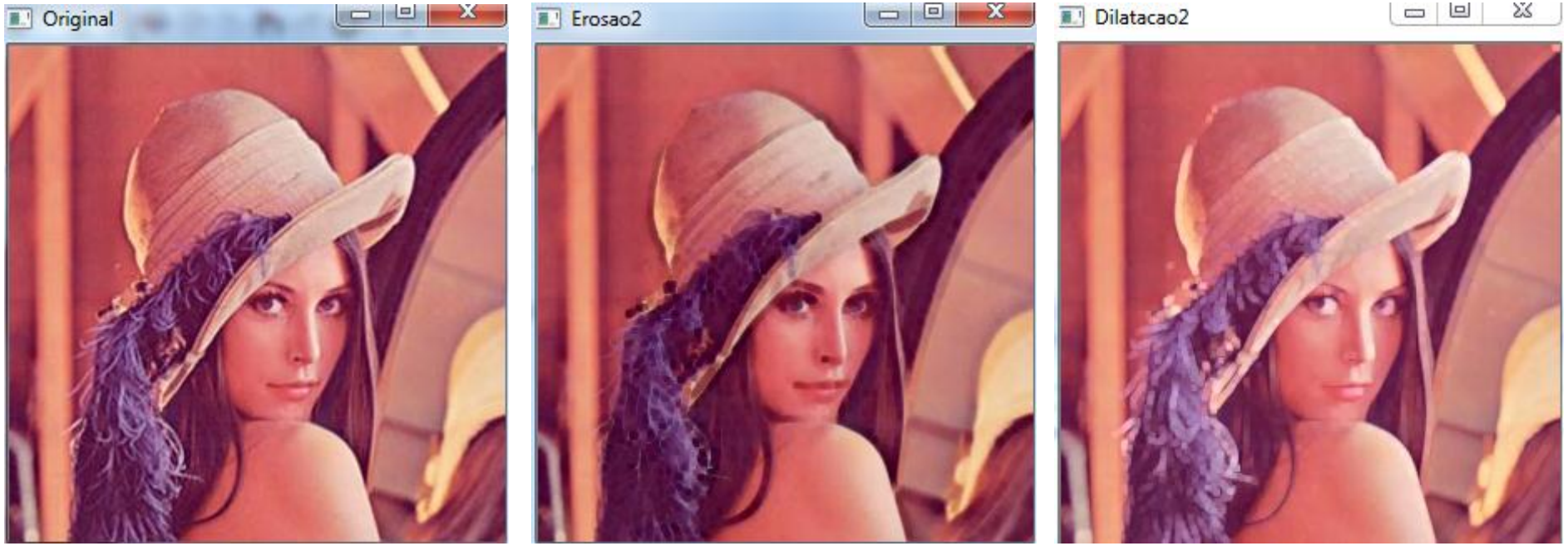






- Elemento estruturante → MORPH_RECT (3x3)
- Aplicado por 10x tanto na erosão quanto na dilatação

Operações Morfológicas - Colorida

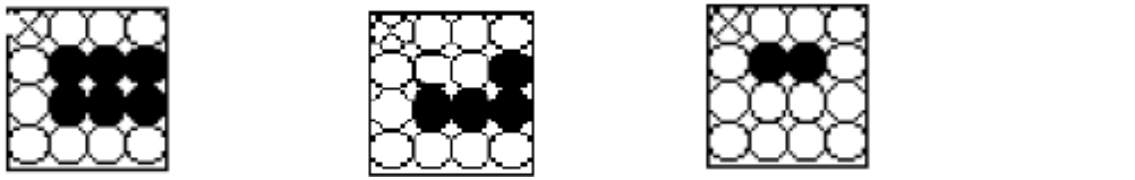


- Elemento estruturante → MORPH_RECT (3x3)
- Aplicado por 1x tanto na erosão quanto na dilatação

Gradiente Morfológico

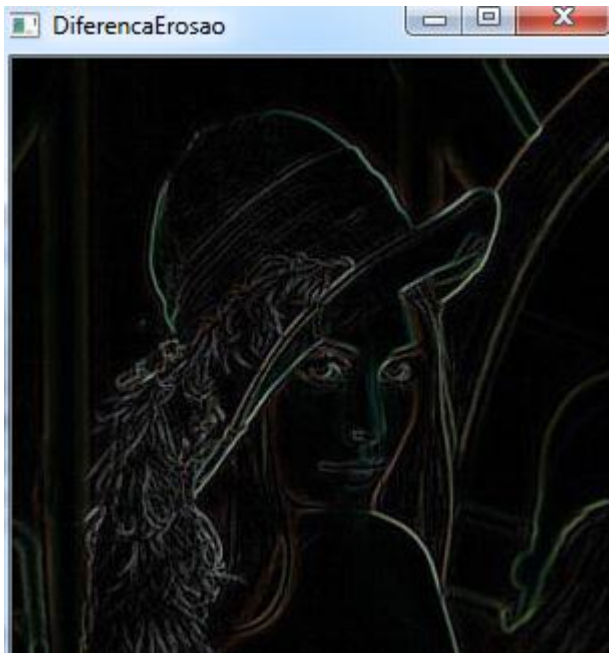
- Esta operação e composta de três outras operações básicas da morfologia: a dilação, erosão e a subtração e é definida da seguinte forma:
- Uma importante aplicação do gradiente é para achar bord $X = (A \oplus B) - (A \ominus B)$

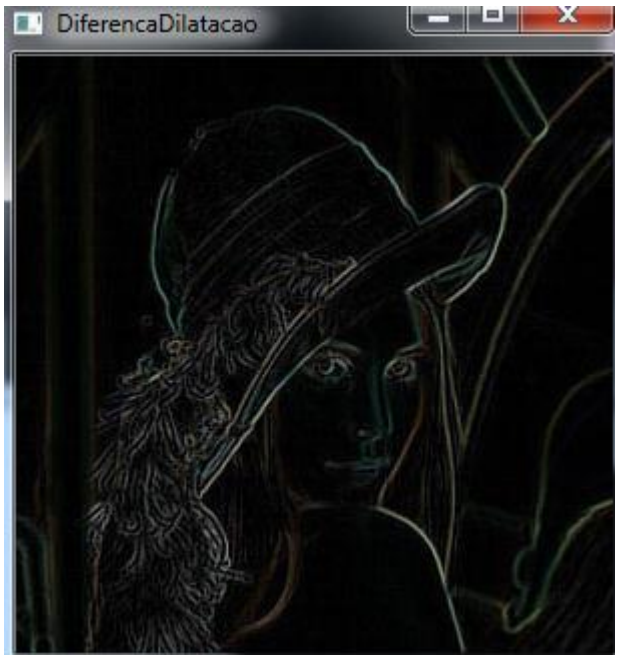
$(A \oplus B)$ $(A \ominus B)$ $X = (A \oplus B) - (A \ominus B)$

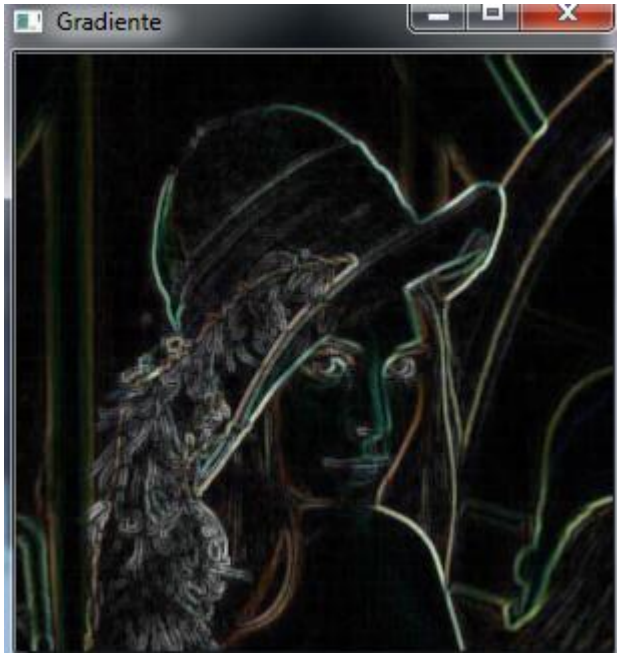


Gradiente Morfológico

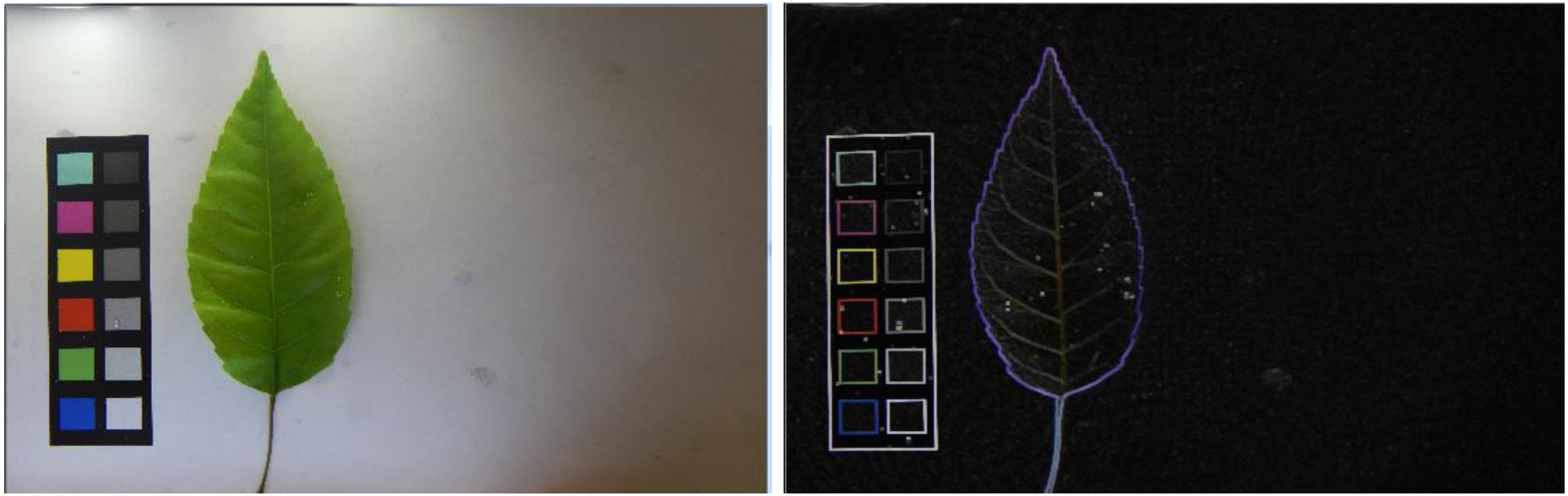
- Operações de subtração são úteis para detecções de bordas. Pode ser por dilatação, erosão ou combinação das duas (gradiente)







Gradiente Morfológico



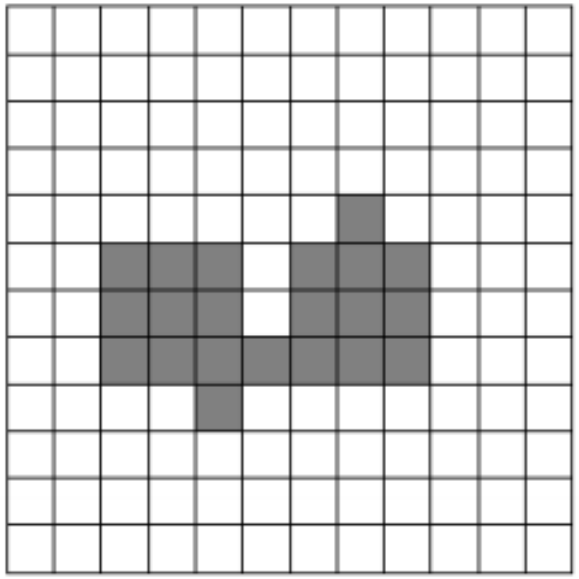
- Elemento estruturante → MORPH_RECT (3x3)
- Aplicado gradiente por 9x

Abertura (opening)

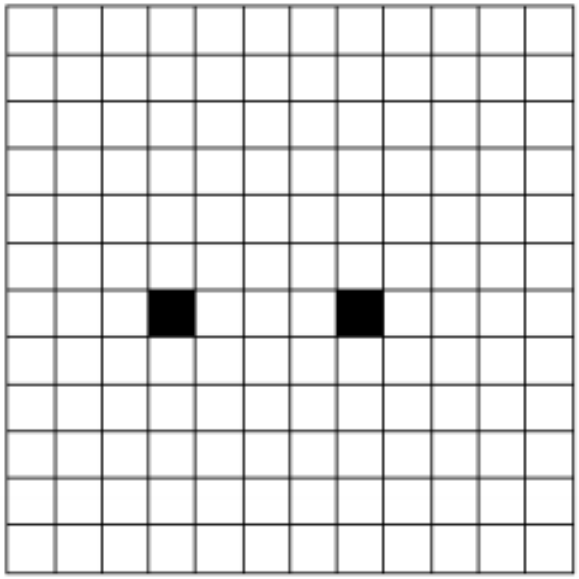
- A abertura em geral suaviza o contorno de uma imagem, quebra estreitos e elimina proeminências delgadas, a operação de abertura é usada também para remover ruídos da imagem.
- Consiste em erodir a imagem A por B e depois dilatar esse conjunto erodido pelo mesmo elemento estruturante.

$$A \bigcirc B = (A \ominus B) \oplus \mathbf{B}$$

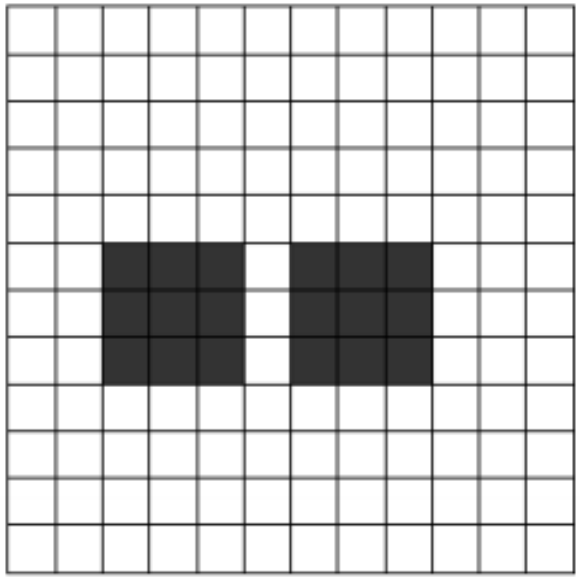
Abertura (opening)



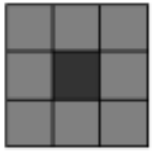
F



$F \ominus H$

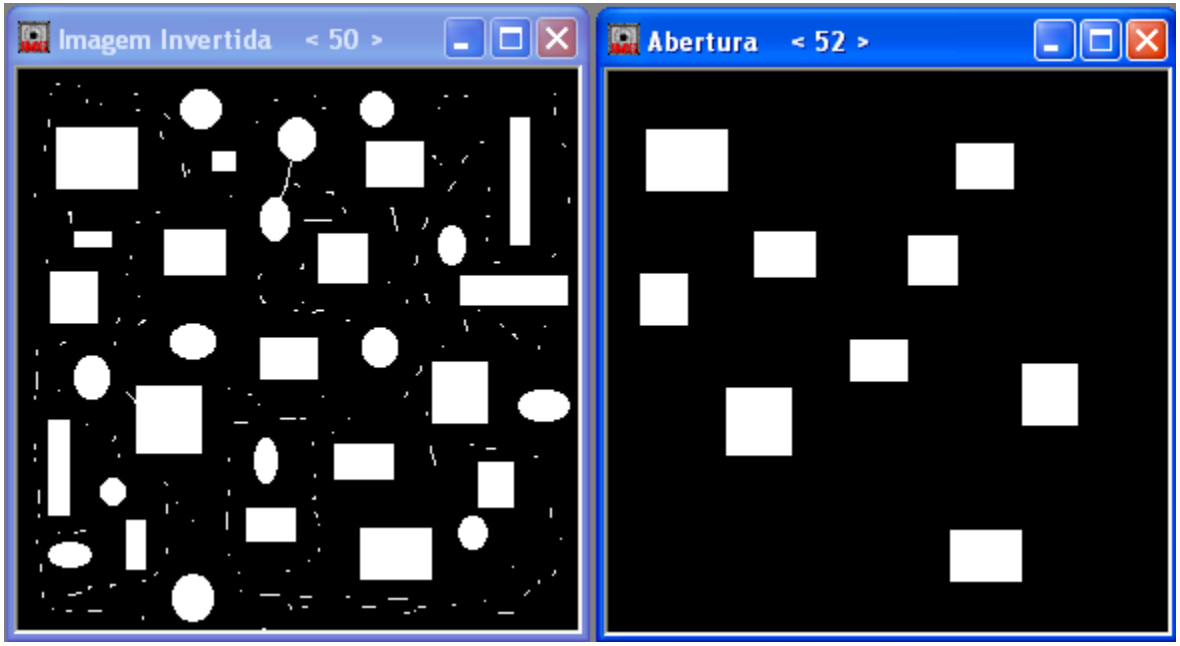


$(F \ominus H) \oplus H$

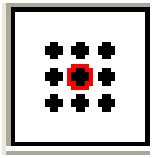


H, 3x3, origin at the center

Abertura



10 x

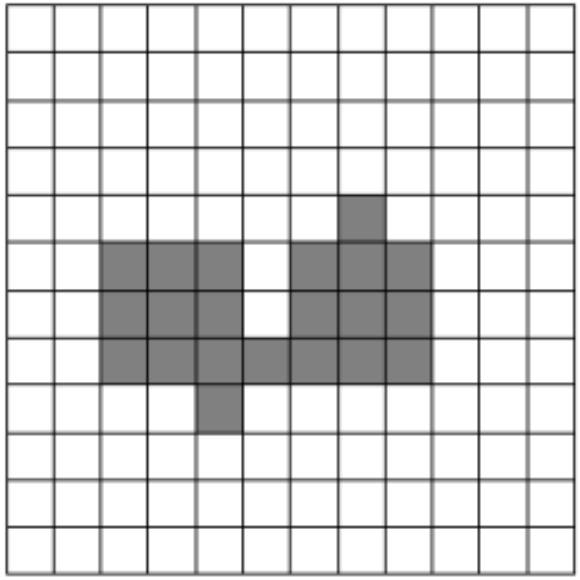


Fechamento (closing)

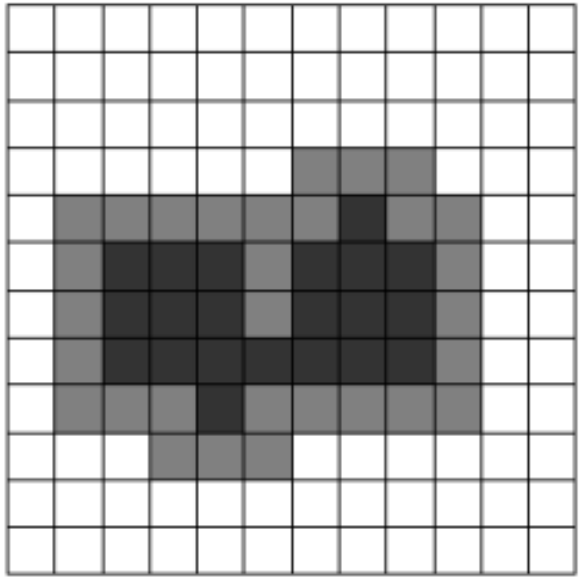
- Suaviza o contorno da imagem, funde pequenas quebras e alarga golfos estreitos, elimina pequenos buracos e preenche fendas de um contorno
- Permite remover muitos dos pixels brancos com ruídos
- O fechamento é definido por:

$$A \bullet B = (A \oplus B) \ominus B$$

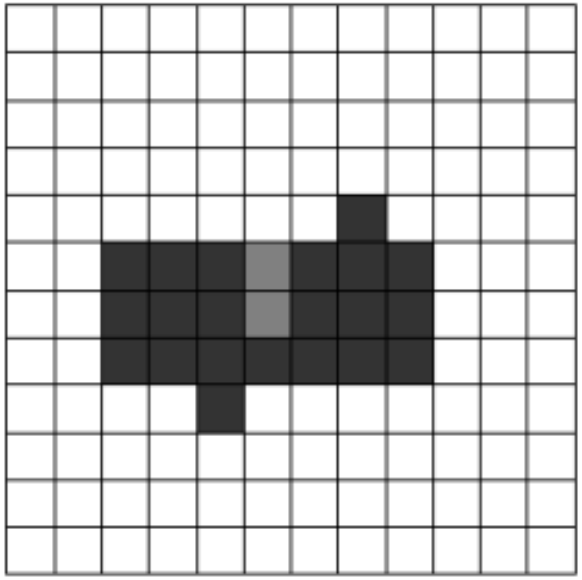
Fechamento (closing)



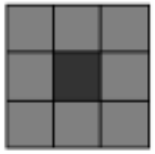
F



$F \oplus H$

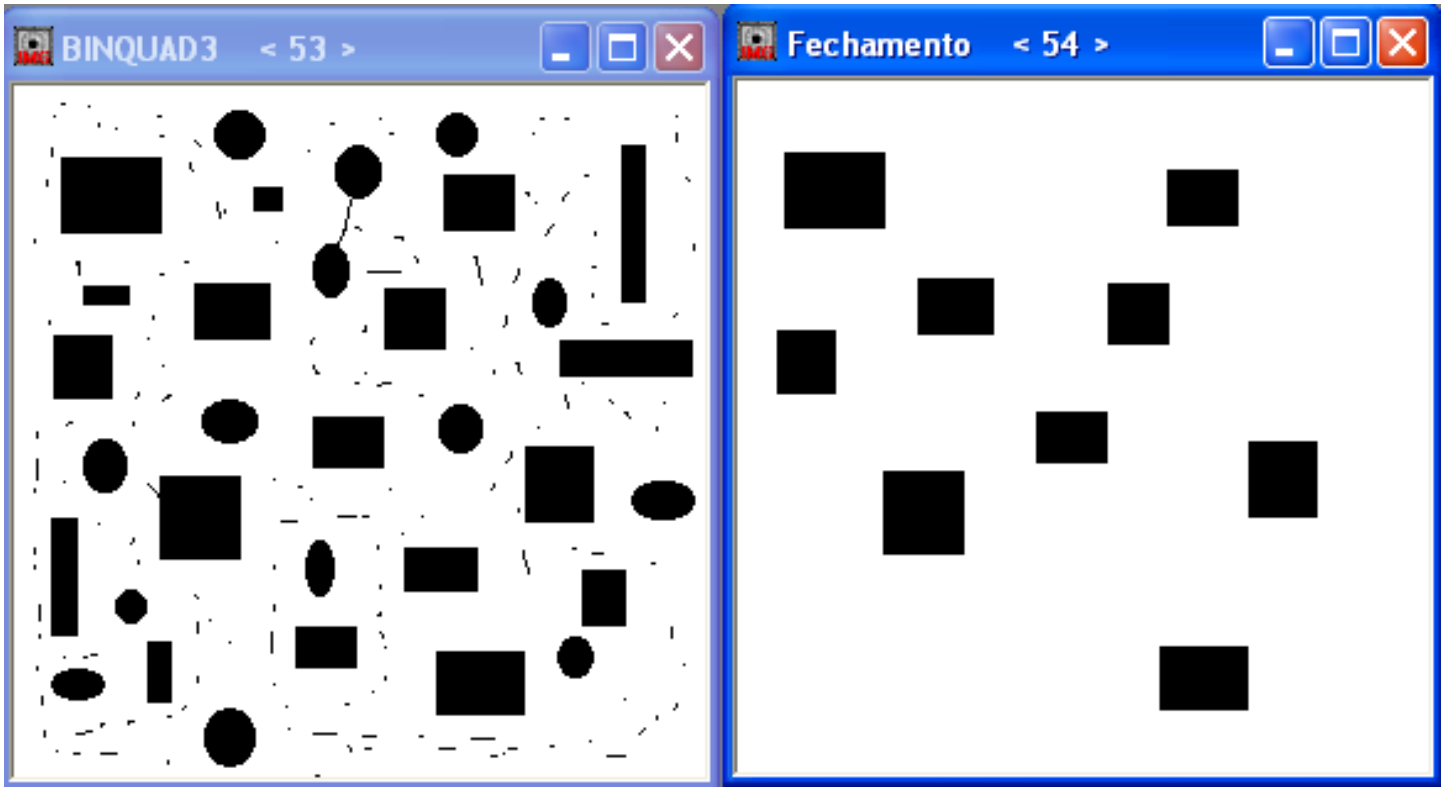


$(F \oplus H) \Theta H$

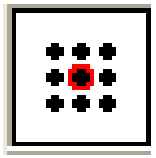


H, 3x3, origin at the center

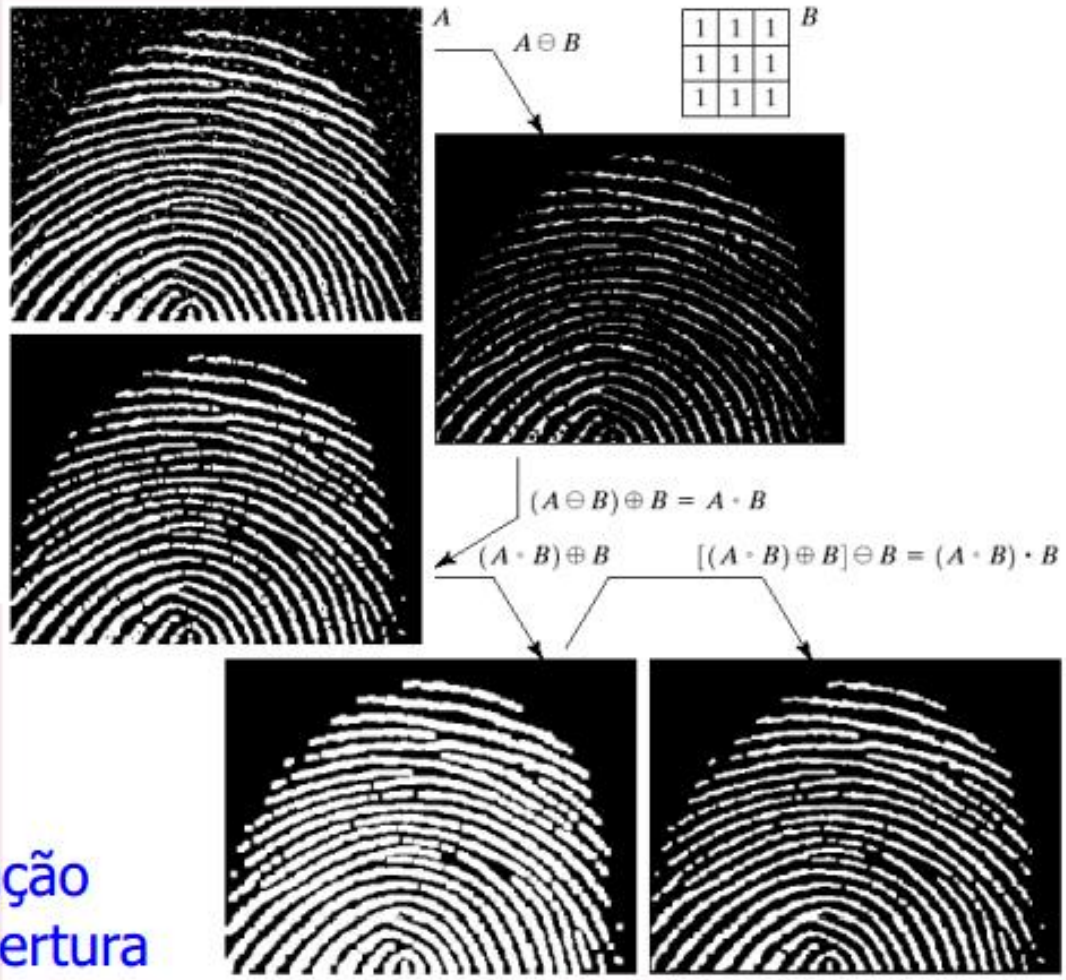
Fechamento (closing)



10 x



Exemplo - Redução de ruído



a b
d c
e f

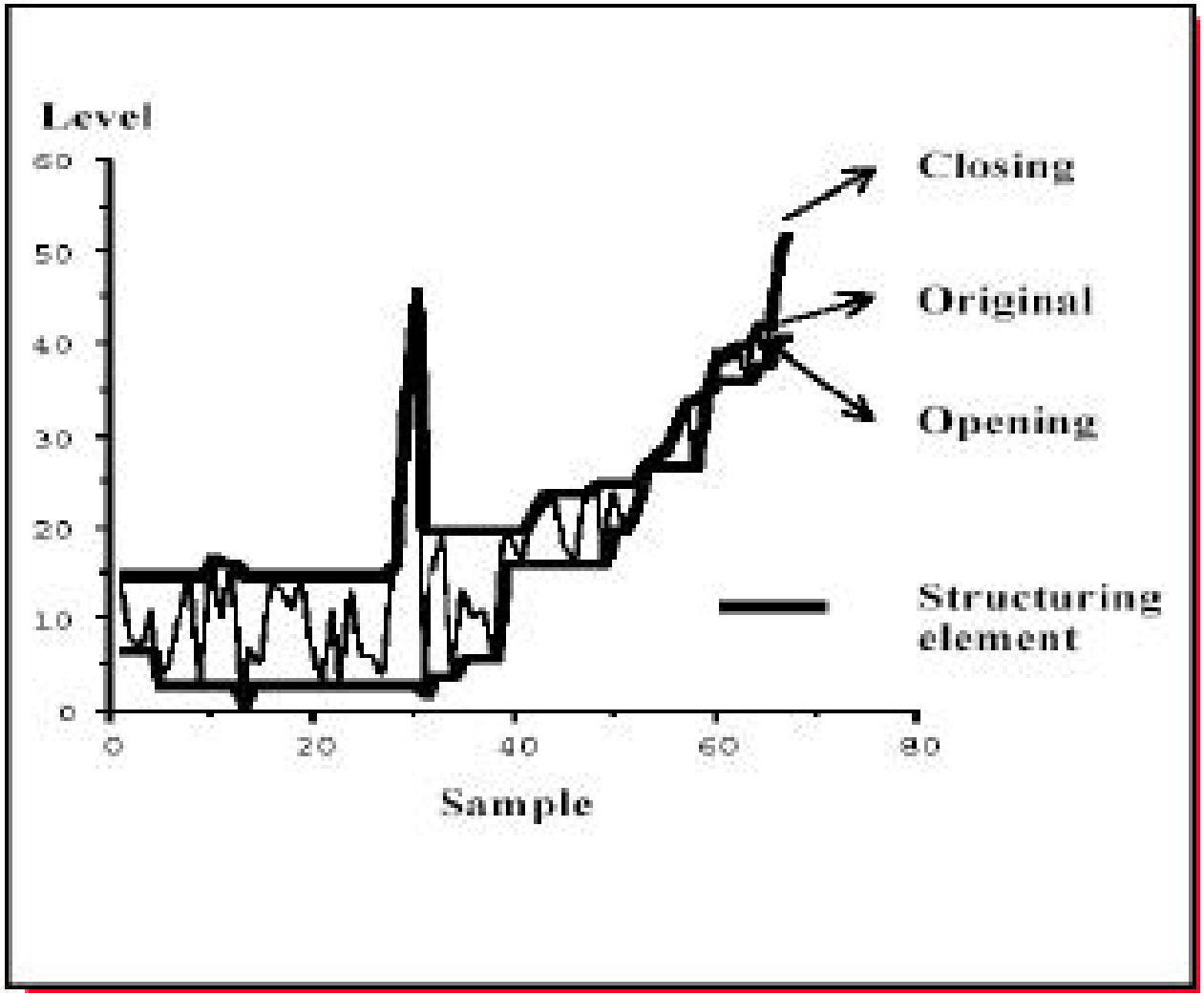
FIGURE 9.11
(a) Noisy image.
(c) Eroded image.
(d) Opening of A.
(d) Dilation of the opening.
(e) Closing of the opening. (Original image for this example courtesy of the National Institute of Standards and Technology.)

Abertura

Dilatação da abertura

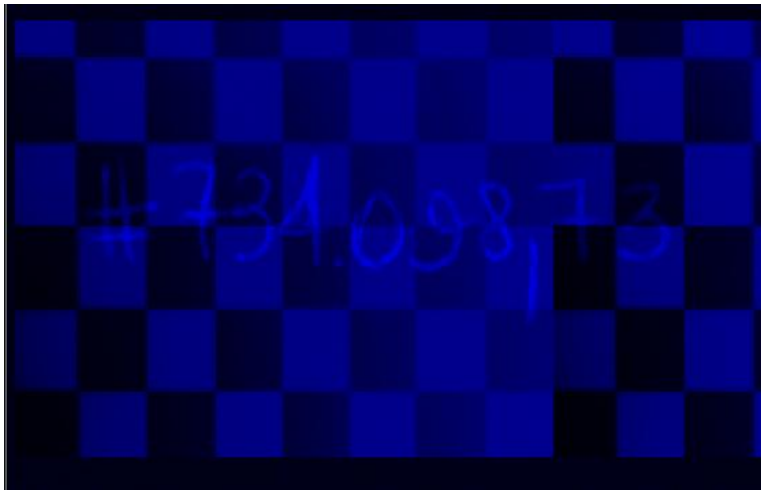
Fechamento da abertura

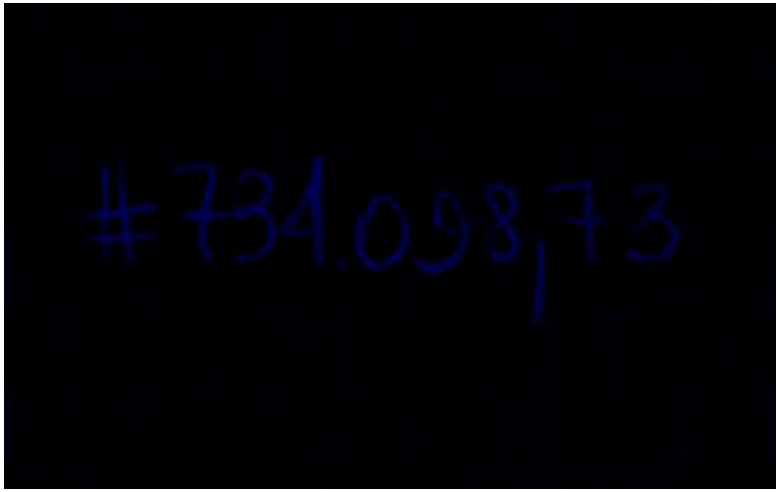
Comparação Abertura-Fechamento



TOP-HAT

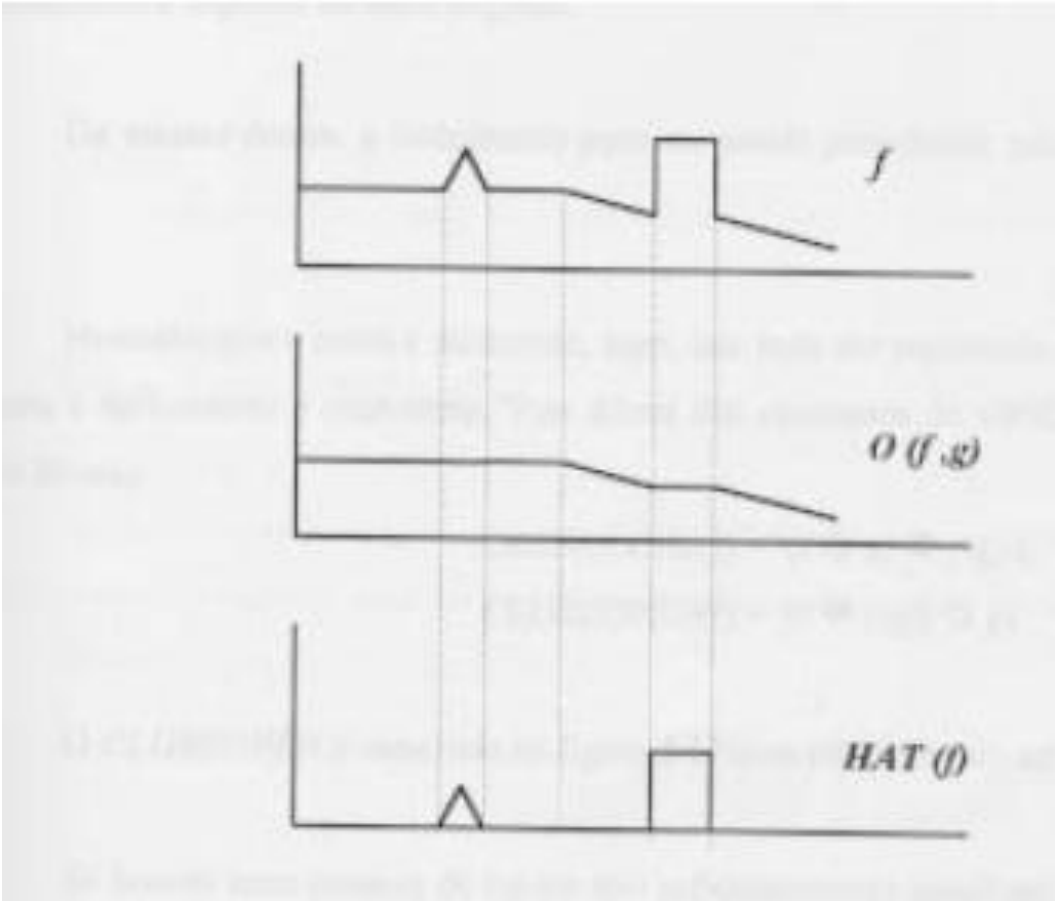
- Subtraindo uma imagem aberta de sua imagem original é possível encontrar pixels escuros em fundo claro.
- O TOP-HAT é usado para encontrar picos
- Para detectar picos: $HAT(f) = f - (f \bigcirc g)$





- Elemento estruturante → MORPH_RECT (3x3) → Aplicado por 7x

TOP-HAT



Original

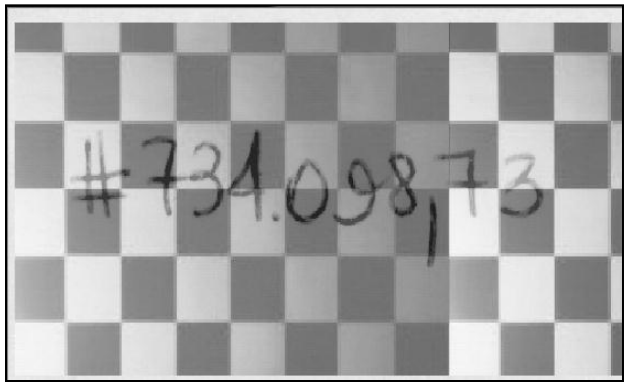
Abertura

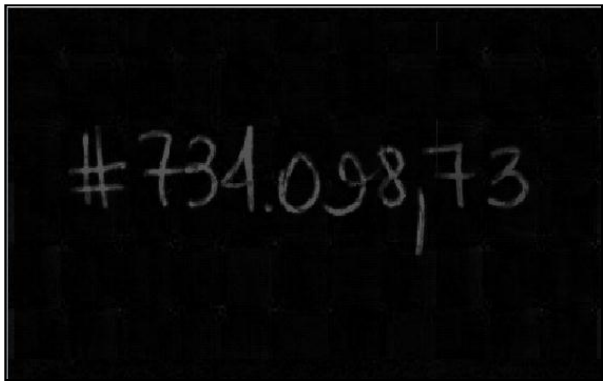
$$HAT(f) = f - (f \circ g)$$

BLACK-HAT

- Variação do TOP-HAT
- Subtrai-se a imagem original da imagem gerada pelo fechamento.
- O BLACK-HAT é usado para encontrar vales de funções.
- Para detectar vales:

$$HAT(f) = (f \bullet g) - f$$





- Elemento estruturante → MORPH_RECT (3x3) → Aplicado por 7x

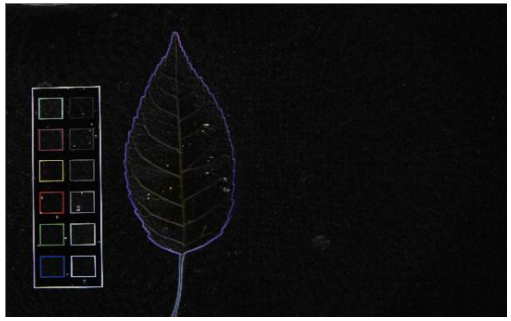
morphologyEx - OpenCv

- Permite transformações morfológicas avançadas
- Sintaxe:
  - void morphologyEx(InputArray src, OutputArray dst, int op, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue())
  - Op → Operador morfológico
    - MORPH_OPEN – abertura
    - MORPH_CLOSE – fechamento
    - MORPH_GRADIENT – gradiente
    - MORPH_TOPHAT – TOP-HAT
    - MORPH_BLACKHAT – BLACK-HAT

morphologyEx - OpenCv

```
Mat image = imread("folha.jpg");
Mat gradiente, topHat, blackHat, abertura, fechamento;
// tipos de EE - MORPH_RECT, MORPH_ELLIPSE, MORPH_CROSS
Mat elemento = getStructuringElement(MORPH_RECT, Size(3,3), Point(-1,-1));

morphologyEx( image, abertura   , MORPH_OPEN     , elemento, Point(-1, -1), 7 );
morphologyEx( image, fechamento, MORPH_CLOSE    , elemento, Point(-1, -1), 7 );
morphologyEx( image, gradiente , MORPH_GRADIENT , elemento, Point(-1, -1), 7 );
morphologyEx( image, topHat     , MORPH_TOPHAT   , elemento, Point(-1, -1), 7 );
morphologyEx( image, blackHat   , MORPH_BLACKHAT , elemento, Point(-1, -1), 7 );
```



FindCountours

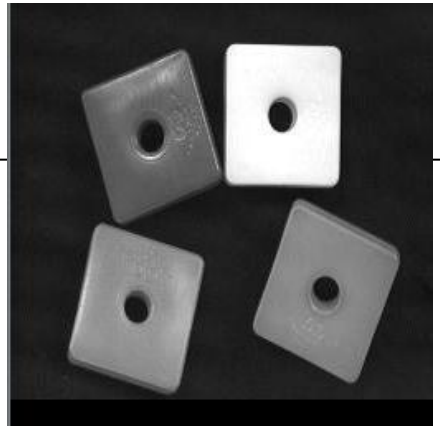
- Encontra contornos em uma imagem binária
- Sintaxe:
  - void findContours(InputOutputArray image, OutputArrayOfArrays contours, OutputArray hierarchy, int mode, int method, Point offset=Point())
  - Contours → Armazena vetor de pontos dos contornos
  - Hierarchy → Vetor opcional com informações hierárquicas da topologia da imagem
  - Mode → Modo de recuperação do contorno
    - CV_RETR_EXTERNAL → retorna só o contorno externos extremos
    - CV_RETR_LIST → retorna todos os contornos sem relacionamento hierárquico
    - CV_RETR_CCOMP → retorna todos os contornos organizados em 2 níveis hierárquicos
    - CV_RETR_TREE → retorna todos contornos e uma reconstrução hierárquica completa
  - Method → Método de aproximação de contorno a ser usado
    - CV_CHAIN_APPROX_NONE → armazena todos os pontos de contorno
    - CV_CHAIN_APPROX_SIMPLE → compacta a saida para elementos unidos
    - CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS

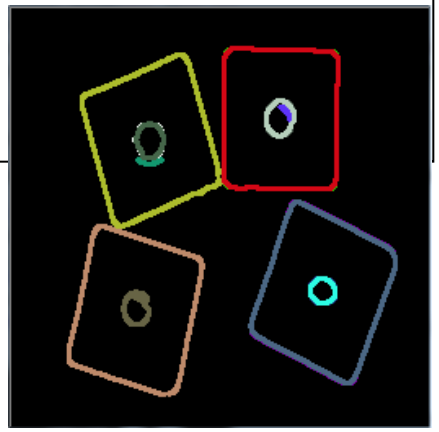
DrawCountours

- Desenha os contornos achados
- Sintaxe:
  - void drawContours(InputOutputArray image, InputArrayOfArrays contours, int contourIdx, const Scalar& color, int thickness=1, int lineType=8, InputArray hierarchy=noArray(), int maxLevel=INT_MAX, Point offset=Point())
  - Contours → vetor de contornos a serem desenhados
  - contourIdx → indica o contorno a ser desenhado (valor negativo → todos)
  - Color → cor a ser usada no desenho
  - Thickness → espessura da linha de desenho
  - lineType → conectividade da linha
  - hierarchy → necessário só se quiser desenhar parte dos contornos
  - maxLevel → nível máximo de hierarquia desejada

```
void thresh_callback(int, void* ){
  Mat canny_output;
  vector<vector<Point> > contours;
  vector<Vec4i> hierarchy;
  Canny( src_gray, canny_output, thresh, thresh*2, 3 );
  findContours( canny_output, contours, hierarchy, CV_RETR_TREE,
                              CV_CHAIN_APPROX_SIMPLE, Point(0, 0) );
  Mat drawing = Mat::zeros( canny_output.size(), CV_8UC3 );
  cout << contours.size() << endl << "Area: ";
  for( int i = 0; i< contours.size(); i++ ){
     cout << "[" << contourArea(contours[i]) << "]";
     Scalar color = Scalar( rng.uniform(0, 255), rng.uniform(0,255), rng.uniform(0,255) );
     drawContours( drawing, contours, i, color, 2, 8, hierarchy, 0, Point() );
  }
  namedWindow( "Contours", 1 );
  imshow( "Contours", drawing );
}
```

```
int main(){
  src = imread("block.bmp");
  cvtColor( src, src_gray, CV_BGR2GRAY );
  blur( src_gray, src_gray, Size(3,3) );
  namedWindow( "Source", 1 );
  imshow( "Source", src );
  createTrackbar( " Canny thresh:", "Source", &thresh, max_thresh, thresh_callback );
  thresh_callback( 0, 0 );
  waitKey(0);
  return(0);
}
```

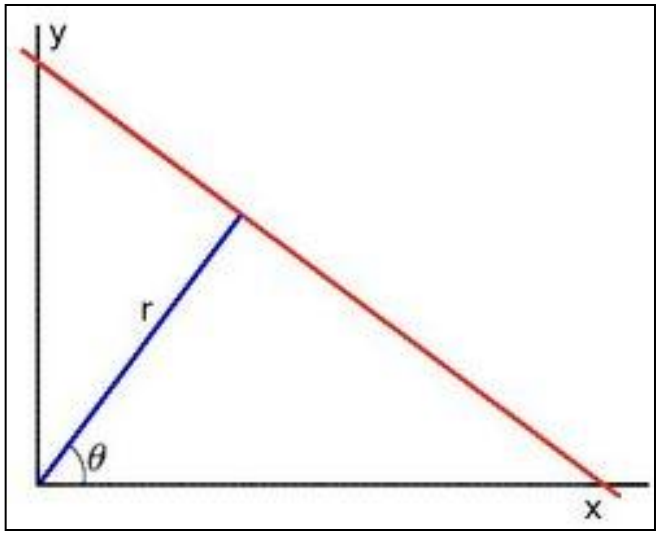




Transformada de Hough - Padrão

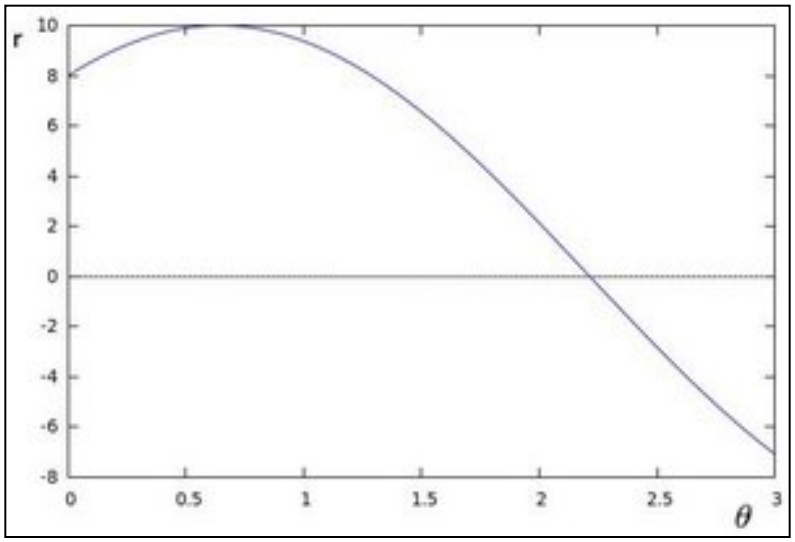
- Método para detecção de formas que são facilmente parametrizadas (linhas, círculos, elipses, etc.), usado, em geral, após a detecção de bordas
- Expressa-se uma linha no sistema Polar

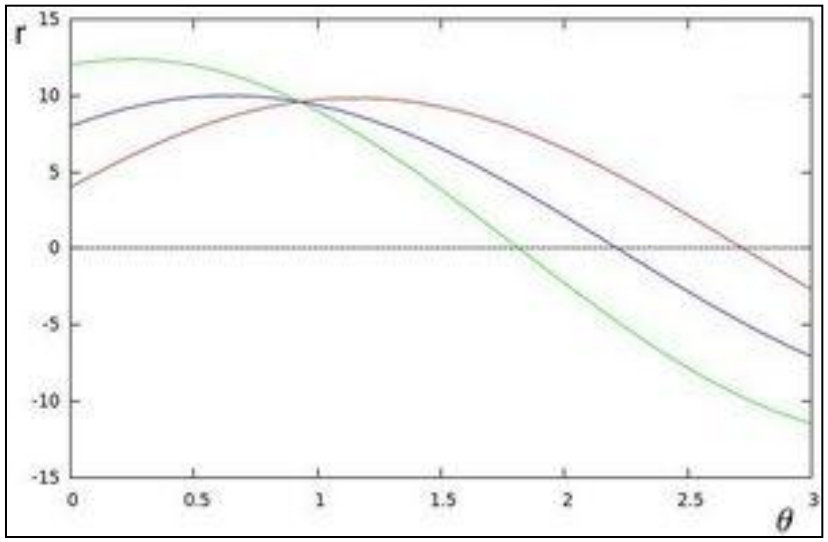
$$r = x \cos\theta + y \sin\theta$$



Transformada de Hough - Padrão

- Para cada ponto (x_0, y_0) é definida uma família de linhas através do calculo de r_0
  - Exemplo: Para o ponto (8,6), gera-se a seguinte senoidal:
  - Fazendo o mesmo para os pontos (4,12) (12,3) tem-se:





Transformada de Hough - Padrão

- Ao se plotar as 3 senoidais em (0.925, 9.6), que representam (θ, r) respectivamente, tem-se a linha que cruza pelos 3 pontos
- Quanto mais curvas intersectando, significa que a linha representada pela intersecção tem mais pontos.
- Partindo-se dessa ideia, calcula-se as curvas para cada ponto da imagem, se o numero de intersecções for acima de um limiar é considerado que existe uma linha cruzando este ponto.

HoughLines – OpenCv - Padrão

- Encontra linhas em uma imagem binária usando a Transformada de Hough
- Sintaxe:

void HoughLines(InputArray image, OutputArray lines, double rho, double theta, int threshold, double srn=0, double stn=0)

  - Lines → vetor de linhas encontrado
  - Rho → distância da origem (r)
  - Theta → ângulo de rotação (θ)
  - Threshold → limiar do acumulador de pontos de intersecção
  - Srn/stn → Usado para Hough multi-escala

Transformada de Hough - Probabilistico

- É uma implementação mais eficiente da Transformada de Hough para linhas
- Sintaxe:

  void HoughLinesP(InputArray image, OutputArray lines, double rho, double theta, int threshold, double minLineLength=0, double maxLineGap=0)

  - minLineLenght → segmentos de linha menor que esse limiar são descartados
  - maxLineGap → separação máxima entre pontos da mesma linha

```
Mat src = imread(filename, CV_LOAD_IMAGE_GRAYSCALE);
Mat dst, cdst, cdstProb;
Canny(src, dst, 50, 200, 3);
cvtColor(dst, cdst, CV_GRAY2BGR);
vector<Vec2f> lines;
HoughLines(dst, lines, 1, CV_PI/180, 80, 0, 0 );
for( size_t i = 0; i < lines.size(); i++ )  {
    float rho = lines[i][0], theta = lines[i][1];
    Point pt1, pt2;
    double a = cos(theta), b = sin(theta);
    double x0 = a*rho, y0 = b*rho;
    pt1.x = cvRound(x0 + 1000*(-b));
    pt1.y = cvRound(y0 + 1000*(a));
    pt2.x = cvRound(x0 - 1000*(-b));
    pt2.y = cvRound(y0 - 1000*(a));
    line( cdst, pt1, pt2, Scalar(0,0,255), 3, CV_AA);
}
```

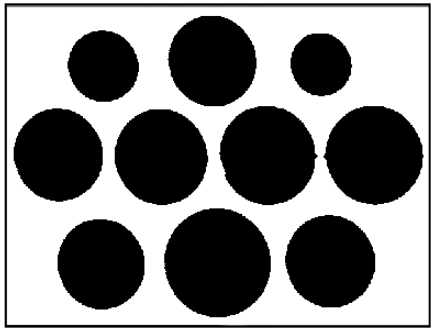
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

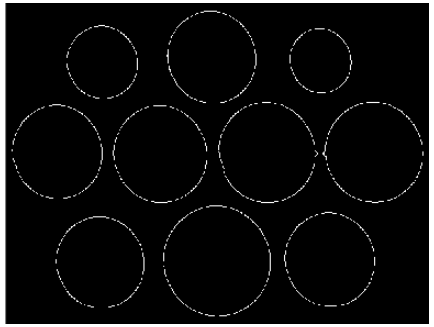
```
vector<Vec4i> linesProb;
HoughLinesP(dst, linesProb, 1, CV_PI/180, 80, 0, 10 );
cvtColor(dst, cdstProb, CV_GRAY2BGR);
for( size_t i = 0; i < linesProb.size(); i++ ){
  Vec4i l = linesProb[i];
  line( cdstProb, Point(l[0], l[1]), Point(l[2], l[3]), Scalar(0,0,255), 3, CV_AA);
}
```

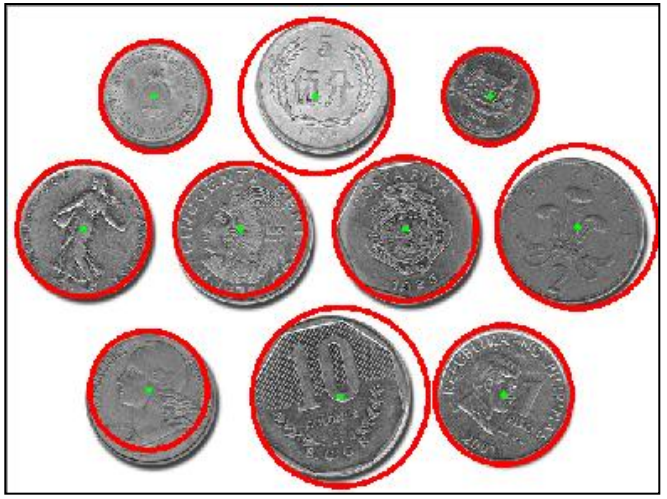
Transformada Hough – Círculos

- Trabalha de forma similar à de linhas, porém com três parâmetros ($x_{central}$, $y_{central}$, raio)
- Encontra círculos em uma imagem em tons de cinza
- Sintaxe:
  - void HoughCircles(InputArray image, OutputArray circles, int method, double dp, double minDist, double param1=100, double param2=100, int minRadius=0, int maxRadius=0)
  - Circles → vetor com os círculos encontrados
  - Method → CV_HOUGH_GRADIENT (único implementado)
  - Dp → Razão inversa da resolução do acumulador x resolução da imagem (dp=1 igual, 2 → metade)
  - minDist → distância mínima de raio
  - Param1 → Threshold máximo para o canny (mínimo → Param1/2)
  - Param2 → Limiar do acumulador Hough
  - minRadius, masRadius → raio mínimo e máximo, respectivamente

```
Mat img, gray, borda, orig;
img=imread("coins.jpg");
orig = img.clone();
cvtColor(img, gray, CV_BGR2GRAY);
threshold(gray, gray, 0, 255, CV_THRESH_BINARY | CV_THRESH_OTSU);
Mat elemento = getStructuringElement(MORPH_ELLIPSE, Size(3,3), Point(-1, -1));
morphologyEx(gray, gray, MORPH_OPEN, elemento, Point(-1, -1), 7);
vector<Vec3f> circles;
Canny(gray, borda, 25, 100, 3);
HoughCircles(borda, circles, CV_HOUGH_GRADIENT, 2, gray.rows/4, 200, 100 );
for( size_t i = 0; i < circles.size(); i++ ){
     Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
     int radius = cvRound(circles[i][2]);
     circle( img, center, 3, Scalar(0,255,0), -1, 8, 0 );
     circle( img, center, radius, Scalar(0,0,255), 3, 8, 0 );
}
```





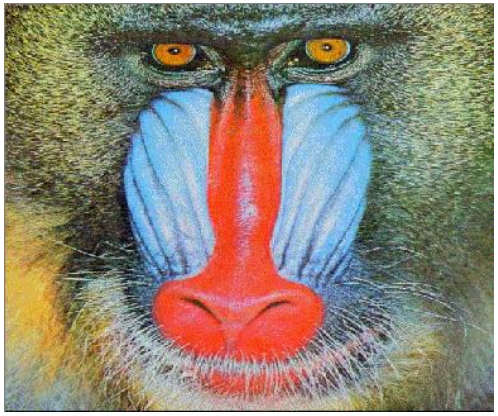


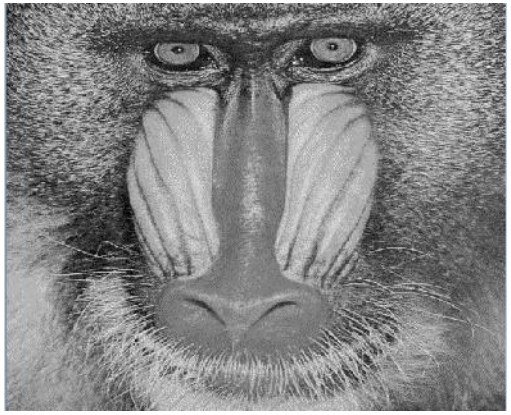
inRange

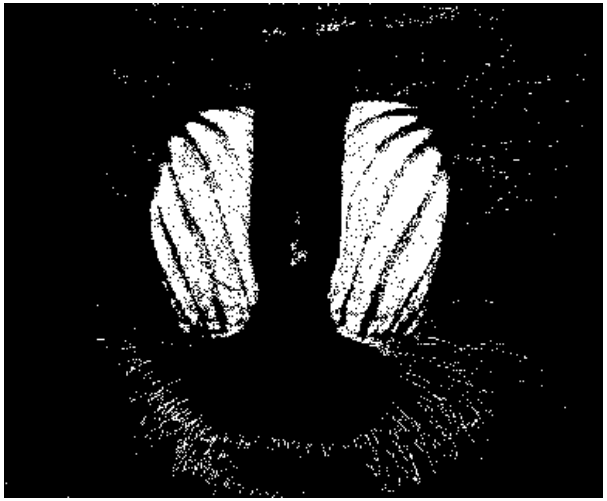
- Verifica se os pixels estão no intervalo entre duas faixas
- Sintaxe: void inRange(InputArray src, InputArray lowerb, InputArray upperb, OutputArray dst)
  - Lowerb → limite baixo (Scalar / valor)
  - Upperb → limite alto (Scalar / valor)
  - Dst → retorna imagem 8 bits

inRange


```
Mat image = imread("baboon.bmp");
Mat binaria, binariaNormal, cinza;

cvtColor (image, cinza, CV_BGR2GRAY);

Scalar min_vals(200, 0, 0);
Scalar max_vals(255, 255, 255);

inRange(image, min_vals, max_vals, binariaNormal);
inRange(cinza, 100, 150, binaria);
```



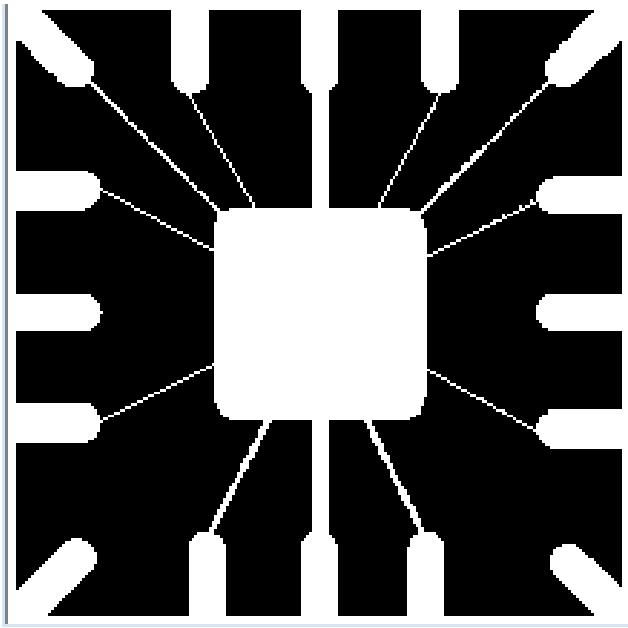


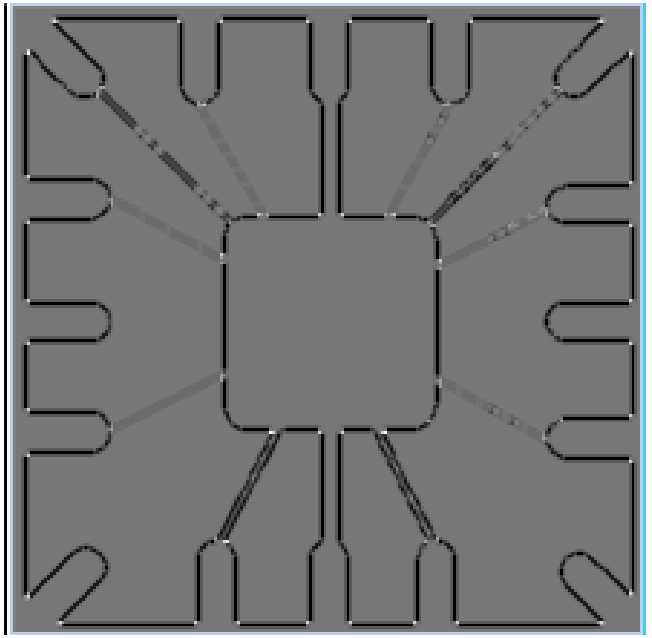


Detector de borda Harris

- Sintaxe: void cornerHarris(InputArray src, OutputArray dst, int blockSize, int ksize, double k, int borderType=BORDER_DEFAULT)
  - src – imagem de entrada 8 bits
  - dst – imagem de saida (CV_32FC1)
  - blockSize – tamanho vizinhança.
  - ksize – Parametro de abertura do Sobel
  - k – Fator de multiplicação do detector Harris
  - borderType – Mtodo para tratar as bordas

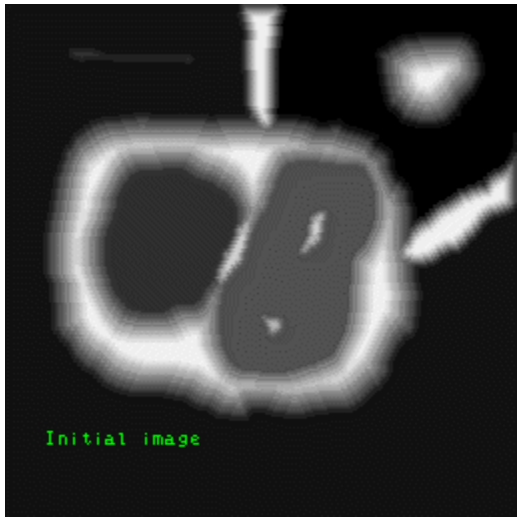
```
Mat original, cinza;
original = imread("linedetection.bmp");
//original = imread("cima.bmp");
cvtColor (original, cinza, CV_BGR2GRAY);

int block_size = 2;
int size_sobel_kernel = 3; // 1, 3, 5 or 7
double k = 0.1;

Mat corners = Mat::zeros(cinza.size(),CV_32FC1);
cornerHarris(cinza, corners, block_size, size_sobel_kernel, k, BORDER_DEFAULT);

normalize(corners, corners, 0, 255, cv::NORM_MINMAX, CV_32FC1, cv::Mat());
cv::convertScaleAbs(corners, corners);
```





Watershed

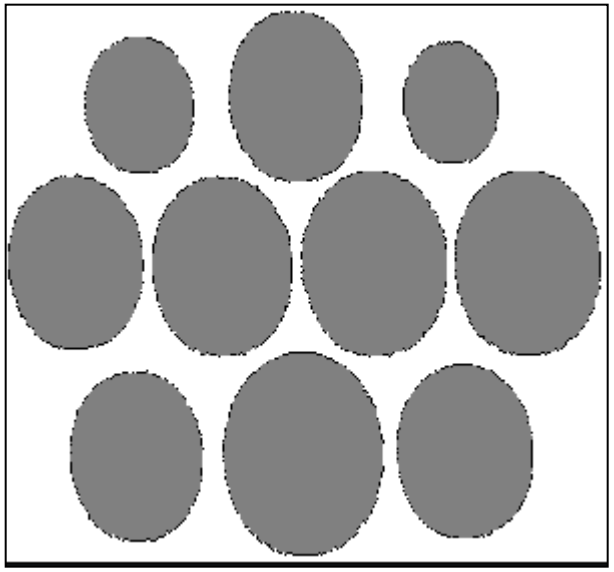
- Uma imagem pode ser vista como uma estrutura topográfica, onde altos tons (claros) aparecem como picos e tons baixos (escuros) como vale, com isso é possivel “encher” os vales



Watershed - OpenCv

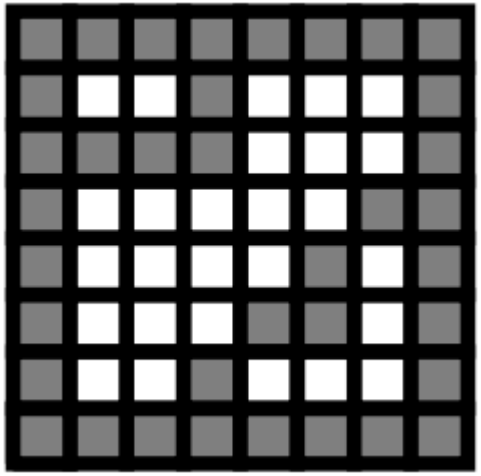
- Faz a segmentação usando o algoritmo watershed
- Sintaxe:
  - void watershed(InputArray image, InputOutputArray markers)

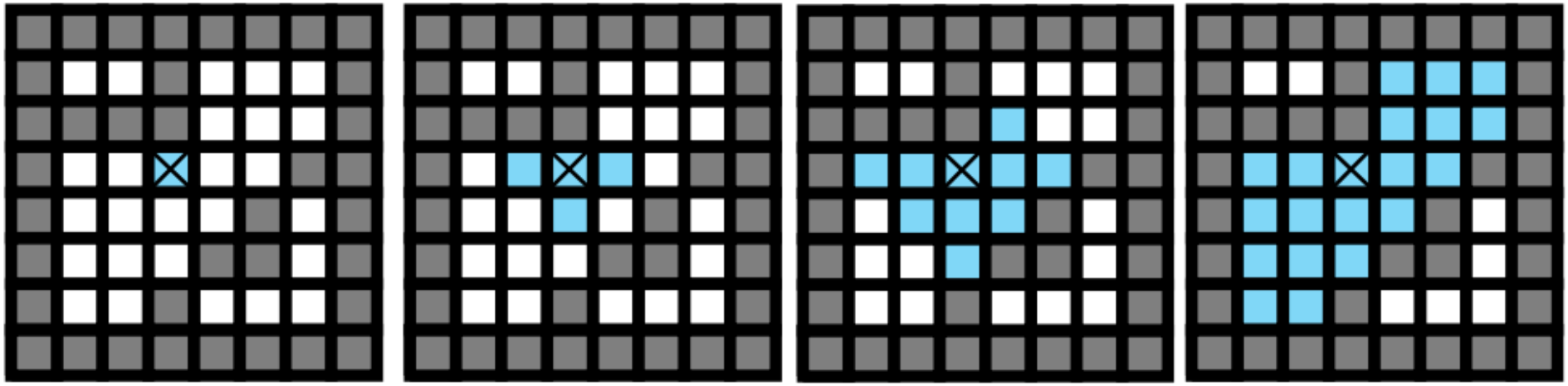
```
Mat image = imread("coins.jpg");
Mat binary, cinza;
cvtColor (image, cinza, CV_BGR2GRAY);
threshold(cinza, binary, 210, 255, THRESH_BINARY);
erode(binary, binary, Mat(), Point(-1, -1), 5);
Mat fg;
fg = binary.clone();
Mat bg;
dilate(binary,bg,Mat(),Point(-1,-1),3);
threshold(bg,bg,1,128,THRESH_BINARY_INV);
Mat markers(binary.size(),CV_8U,Scalar(0));
markers= fg+bg;
markers.convertTo(markers, CV_32S);
watershed(image,markers);
markers.convertTo(markers,CV_8U);
```



Flood Fill

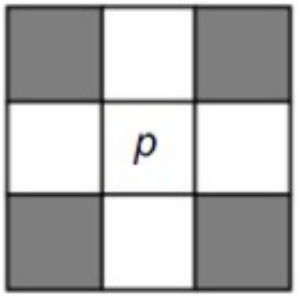
- Preenche pixels conectados dada uma semente

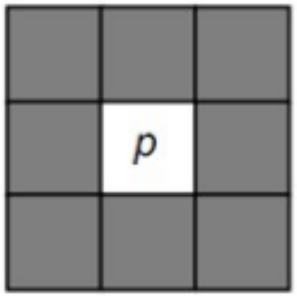




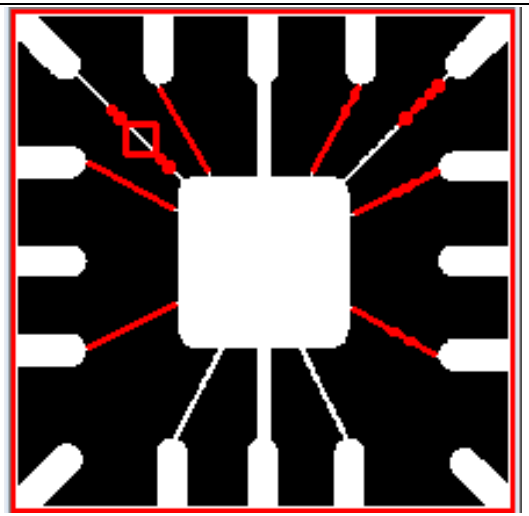
FloodFill - OpenCv

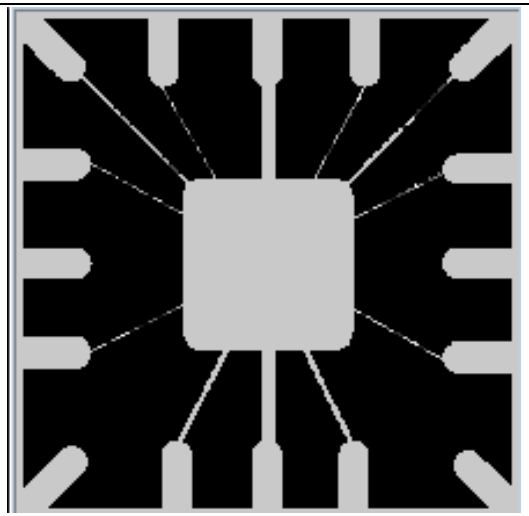
- Sintaxe:
  - int floodFill(InputOutputArray image, Point seedPoint, Scalar newVal, Rect* rect=0, Scalar loDiff=Scalar(), Scalar upDiff=Scalar(), int flags=4)
  - Image → imagem tanto de entrada quanto de saida
  - seedPoint → semente
  - newVal → nova cor a ser usada
  - Rect → tamanho mínimo do retângulo a ser pintado
  - loDiff → máxima diferença para baixo em relação a semente
  - upDiff → máxima diferença para cima em relação a semente
  - Flags → conectividade 4 ou 8



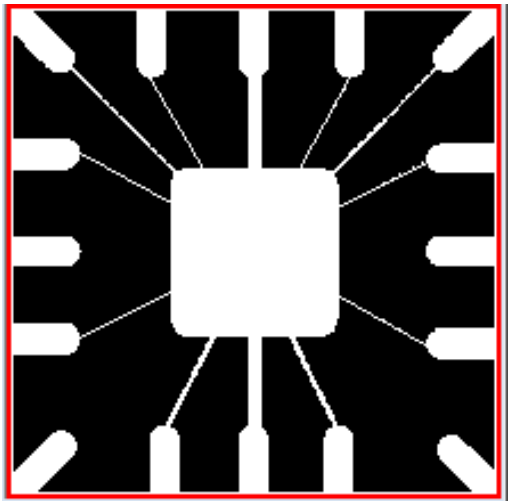


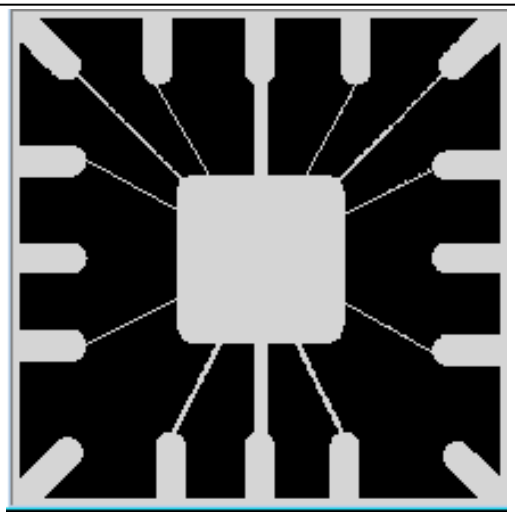
```
int main(){
    Mat original, cinza;
    srand (time(NULL));
    original = imread("linedetection.bmp");
//    original = imread("cima.bmp");
    cvtColor (original, cinza, CV_BGR2GRAY);
    for(int y = 0; y < cinza.rows; y++) {
        for(int x = 0; x < cinza.cols; x++) {
            if (cinza.at<unsigned char>(y,x) == 255) {
                Rect rect;
//                floodFill(cinza, Point(x,y), Scalar(rand()%255), &rect,
//                          Scalar(0), Scalar(0), 8); // conexao 8
                floodFill(cinza, Point(x,y), Scalar(rand()%255), &rect,
                          Scalar(0), Scalar(0), 4); // conexao 4
//                cout << "[" << rect << "]";
                rectangle(original, rect, Scalar(0,0,255), 2, 8, 0);
            }
        }
    }
```





```
int main(){
    Mat original, cinza;
    srand (time(NULL));
    original = imread("linedetection.bmp");
//    original = imread("cima.bmp");
    cvtColor (original, cinza, CV_BGR2GRAY);
    for(int y = 0; y < cinza.rows; y++) {
        for(int x = 0; x < cinza.cols; x++) {
            if (cinza.at<unsigned char>(y,x) == 255) {
                Rect rect;
                floodFill(cinza, Point(x,y), Scalar(rand()%255), &rect,
                          Scalar(0), Scalar(0), 8); // conexao 8
//                floodFill(cinza, Point(x,y), Scalar(rand()%255), &rect,
//                          Scalar(0), Scalar(0), 4); // conexao 4
//                cout << "[" << rect << "]";
                rectangle(original, rect, Scalar(0,0,255), 2, 8, 0);
            }
        }
    }
```





Filtro de ruído

- Sal
  - Gera pixels brancos
- Pimenta
  - Gera pixels pretos

```
void salt(Mat &image, int n) {
   for (int k=0; k<n; k++) {
      int i= rand()%image.cols;
      int j= rand()%image.rows;
      if (image.channels() == 1) { // gray-level image
         image.at<uchar>(j,i)= 255;
      } else if (image.channels() == 3) { // color image
         image.at<cv::Vec3b>(j,i)[0]= 255;
         image.at<cv::Vec3b>(j,i)[1]= 255;
         image.at<cv::Vec3b>(j,i)[2]= 255;
      }
   }
}
```

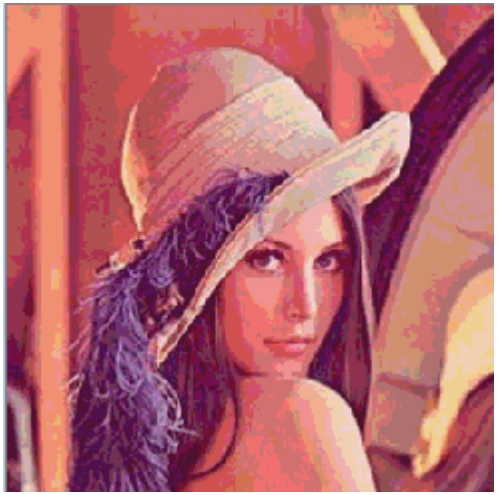
Redução de Cor

```
void colorReduce(Mat &image, int div=64) {
      int nl= image.rows;
      int nc= image.cols * image.channels();
      for (int j=0; j<nl; j++) {
         uchar* data= image.ptr<uchar>(j);
         for (int i=0; i<nc; i++) {
             data[i]=    data[i]/div*div + div/2;
         }
      }
}
```

128

64

32



Seleção de cores semelhantes

```
int main(){
   char tecla;
   src = imread("lena.bmp");
   dst = src.clone();
   namedWindow("orig", 0);
   namedWindow("dst", 0);
   cvSetMouseCallback("orig", mouseEvent, &src);
   do{
     imshow("orig", src);
     imshow("dst", dst);
     tecla = waitKey(5);
   } while (tecla != 27);
   return 0;
}
```

```
void mouseEvent(int evt, int x, int y, int flags, void* param) {
    Mat *aux = (Mat*) param;
    if (evt == CV_EVENT_LBUTTONDOWN) {
       cout << (*aux).at<Vec3b>(y, x);
       dst = filtraCorSelecionada((*aux).at<Vec3b>(y,x));
    }
}
```

```
int getDistance(const Vec3b& color, const Vec3b& target) {
        return abs(color[0]-target[0])+
               abs(color[1]-target[1])+
               abs(color[2]-target[2]);
}
```

```
Mat filtraCorSelecionada(const Vec3b& cor) {
     Mat result;
     result.create(src.rows,src.cols,CV_8U);
     Mat_<Vec3b>::const_iterator it    = src.begin<Vec3b>();
     Mat_<Vec3b>::const_iterator itend = src.end<Vec3b>();
     Mat_<uchar>::iterator itout       = result.begin<uchar>();
     for ( ; it!= itend; ++it, ++itout) {
        if (getDistance(*it, cor)<minDist)
           *itout= 255;
        else
           *itout= 0;
     }
     return result;
}
```

Exercício 1

- Contem quantas galinhas existem no retângulo indicado, mostre o passo-a-passo

Exercício 2

- Crie uma imagem binária que apresente todas as bolas, mostre o passo-a-passo