Transformada de Hough para detectar retas.

Vamos estudar transformada de Hough, para aprender como funciona, e não apenas chamar a função pronta do OpenCV. As transformadas de Hough do OpenCV possuem várias limitações.

Como detectar a reta na figura abaixo?

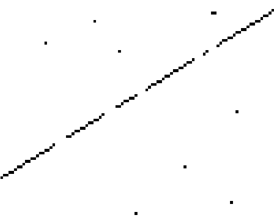


Figura 1 (reta.bmp)

A transformada de Hough é capaz de detectar grupos de pixels que pertencem a uma linha reta (mesmo que esteja quebrada e/ou com ruídos). Uma linha reta é geralmente descrita como $y = mx + b$. As características desta reta são a inclinação m e a intersecção b. Assim, uma reta $y = mx + b$ pode ser representada como um ponto (b, m) no espaço dos parâmetros.

Porém, ambos parâmetros são ilimitados, isto é, à medida em que a reta torna-se vertical, as magnitudes de b e m tendem ao infinito. Assim, para efeitos computacionais, é melhor parametrizar as retas usando dois outros parâmetros (θ,ρ): $x \cos\theta + y\, sen\,\theta = \rho$. Veja a figura 2. Isto associa cada reta da imagem a um único ponto (θ,ρ) no plano dos parâmetros (ou espaço de Hough).

Infinitas retas passam por um ponto no plano. Todas as retas que passam por esse ponto formam um senóide no plano de Hough (figuras 3a e 3b).

Dois pontos p e q no plano da imagem definem uma reta pq. Dois pontos p e q correspondem a dois senóides no plano de Hough. A intersecção dos dois senóides representa a reta pq que passa pelos dois pontos p e q no plano da imagem (figuras 3c e 3d).

Uma reta no plano da imagem corresponde a infinitos senóides no plano de Hough que intersectam num único ponto (figuras 3e e 3f). Este ponto do plano de Hough representa a reta (isto é, os parâmetros deste ponto no espaço de Hough representam os parâmetros da reta na imagem original).

4 retas no plano da imagem correspondem a 4 senóides no plano de Hough que se acumulam em 4 pontos (figuras 3g e 3h). Note que os 2 pontos de acumulação nas bordas esquerda e direita da figura 3h correspondem a uma única reta.

É possível melhorar consideravelmente a velocidade de processamento utilizando a informação sobre orientação local das arestas (normalmente através do gradiente).

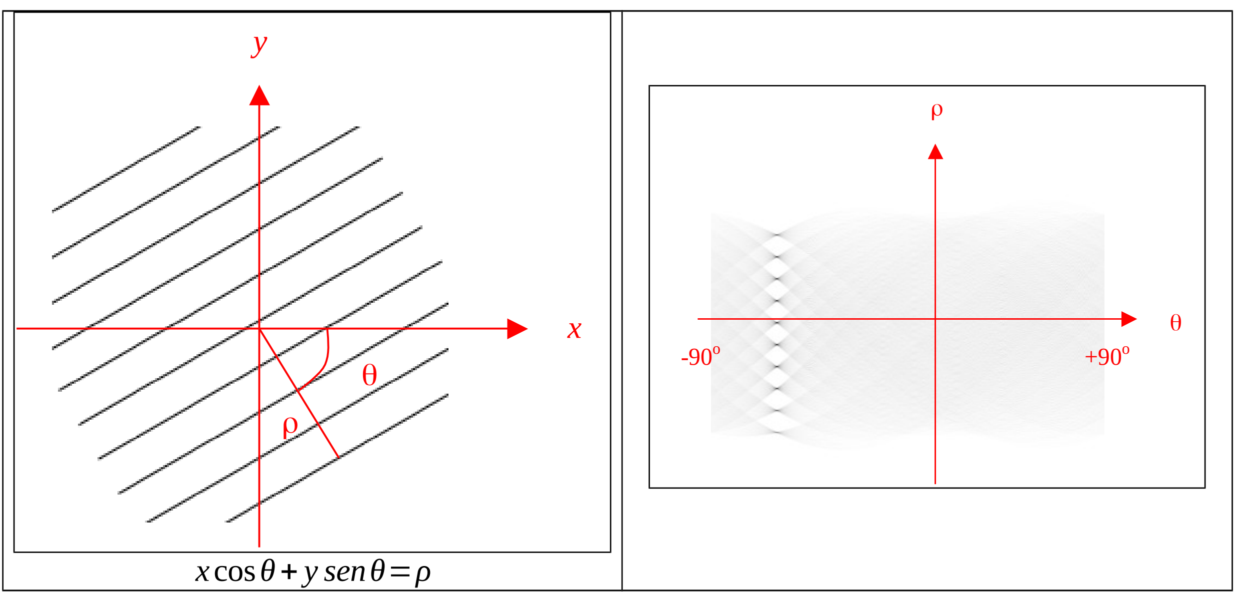


Figura 2: Parâmetros da transformada de Hough no espaço (θ,ρ).

Aplicação: Corrigir automaticamente a rotação de um documento escaneado.

Nota: Em imagens em níveis de cinza, é possível usar gradiente para estimar θ e com isso acelerar o processamento.

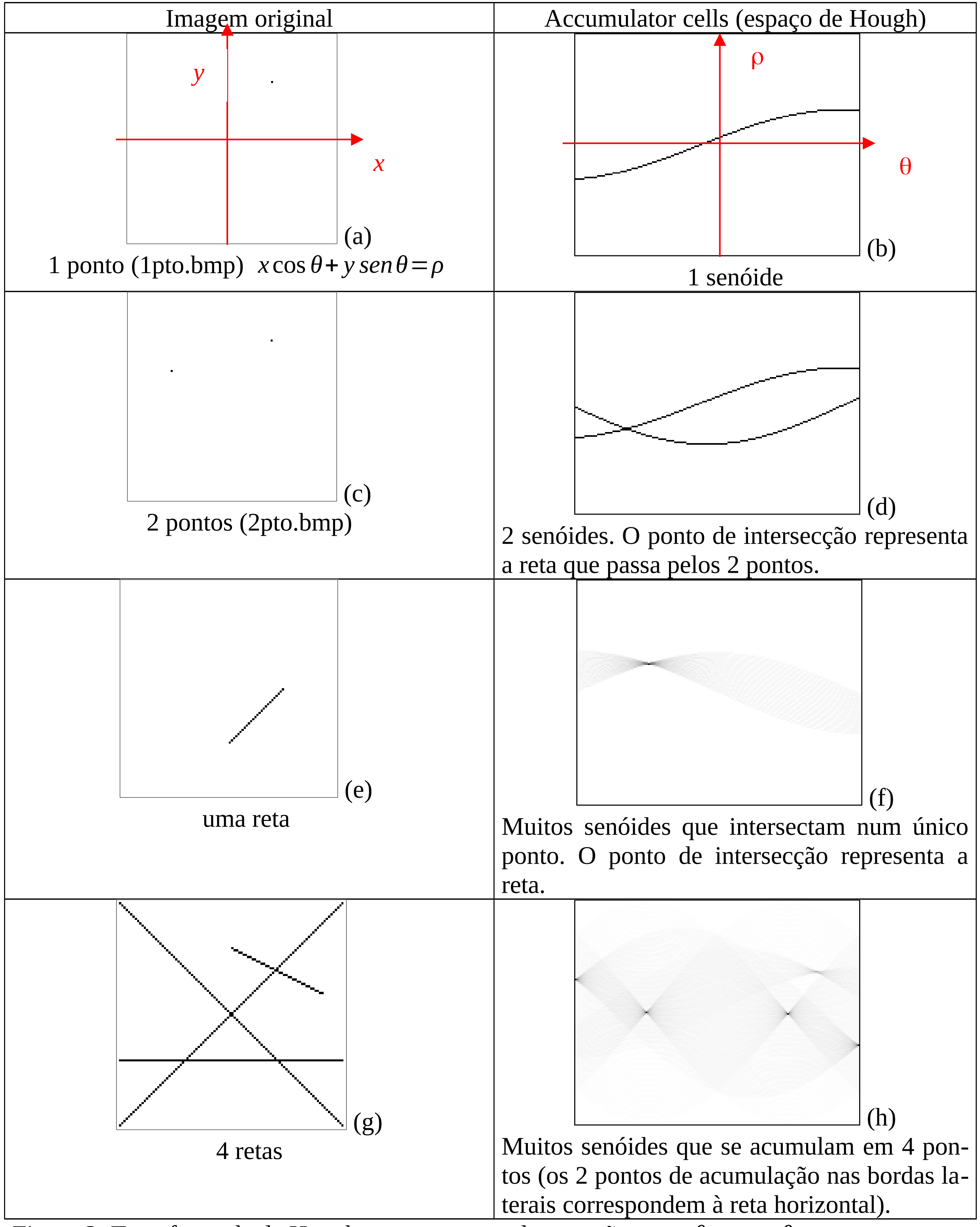
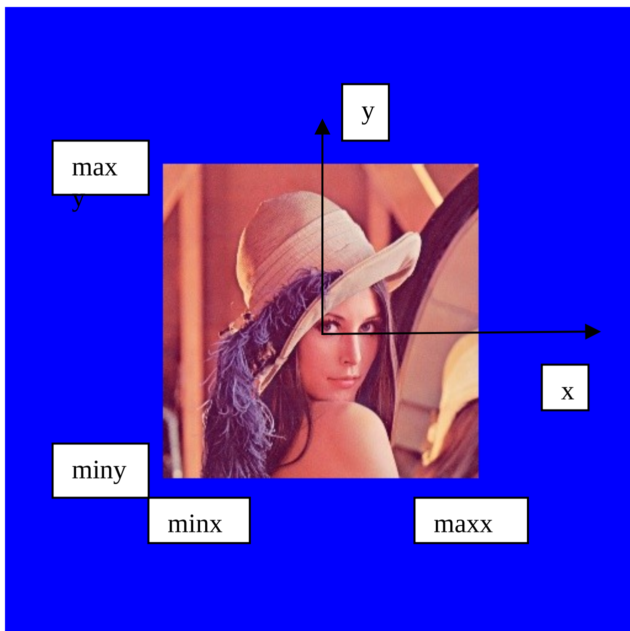


Figura 3: Transformada de Hough para retas usando equação $x\cos\theta + y\,sen\theta = \rho$.

Como acessar uma imagem com coordenadas (x,y) com origem no centro da imagem? Além disso, se acessar fora do domínio, queremos que devolva uma cor de background.

Na biblioteca Cekeikon, há a classe `ImgXyb<T>` derivada da classe `Mat_<T>`.



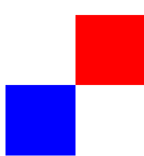
ImgXyb<GRY> im é igual em tudo a Mat_<GRY> ma, exceto que acessa os pixels em ordem (x,y).

Além disso, ImgXyb<GRY> im possui:

```
im.centro(l,c); //Define o centro da imagem
im.lc; // Devolve a linha central - nao serve para definir linha central
im.cc; // Devolve a coluna central - nao serve para definir coluna central
im.minx; // Menor coordenada x da imagem
im.maxx; // Maior coordenada x da imagem
im.miny; // Menor coordenada y da imagem
im.maxy; // Maior coordenada y da imagem
im.backg; // Cor de fundo. Usada tanto para definir (set) a cor de fundo
          // como para consultar (get) a cor de fundo
```

Programa simples para testar funcionalidade de ImgXyb:

```
//imgxyb.cpp
#include <cekeikon.h>
int main() {
  ImgXyb<COR> a(3,3, COR(255,255,255));
  a.centro(1,1);
  a.backg=COR(0,0,255);
  a(-1,-1)=COR(255,0,0);
  a(0,0)=a(-100,100);
  imp(a,"imgxyb.png");
}
```



ImgXyb<T> im é uma classe derivada de Mat_<T> ma. Pode-se usar im nas funções que se espera receber ma. Se houver função especial para im, esta será chamada (em vez da versão genérica para ma).

Programa Hough simplificado para detectar retas em imagens binárias 100x100. O programa abaixo só funciona para imagens 100x100 para simplificar o programa por motivos didáticos.

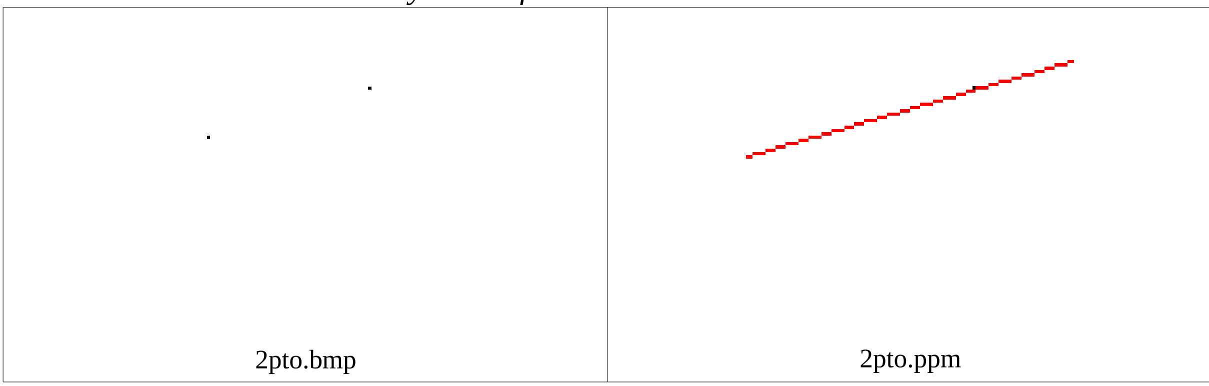
```
//houghsimp.cpp - pos2017
#include <cekeikon.h>

int main() {
   ImgXyb<GRY> ent; le(ent,"reta.bmp");
   ent.centro(ent.rows/2,ent.cols/2);

   int nl2=teto( sqrt(50*50+50*50) );
   int nl=2*nl2+1;
   int nc2=90;
   int nc=2*nc2+1;

   ImgXyb<FLT> sai(nl,nc,1.0);
   sai.centro(nl2,nc2);

   for (int xa=ent.minx; xa<=ent.maxx; xa++) {
      for (int ya=ent.miny; ya<=ent.maxy; ya++) {
         if (ent(xa,ya)==0) {
            for (int theta=-90; theta<=+90; theta++) {
               double rad=deg2rad(theta);
               int rho=cvRound((xa*cos(rad)+ya*sin(rad)));
               sai(theta,rho) -= 1;
            }
         }
      }
   }
   sai=normaliza(sai);
   imp(sai,"reta-ho.png");
}
```

(PSI3472-2019 Aula 3 Exercício 1) Adapte houghsimp.cpp para aceitar como parâmetros imagens de entrada e saída. Execute houghsimp.cpp para imagens 1pto.bmp, 2pto.bmp e reta.bmp:

```
> exercicio1 1pto.bmp 1pto.ppm
> exercicio1 2pto.bmp 2pto.ppm
> exercicio1 reta.bmp reta.ppm
```

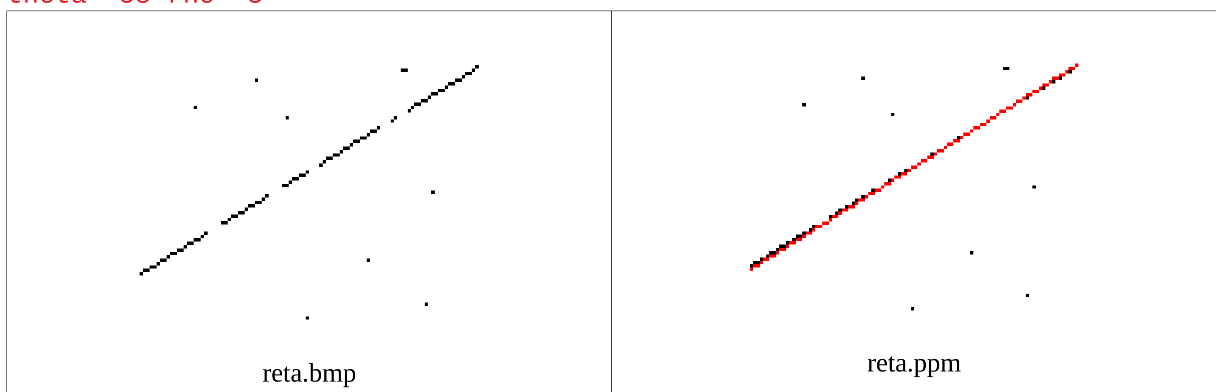
(PSI3472-2019 Aula 3 Exercício 2) Modifique o programa houghsimp.cpp para que detecte a reta-suporte na imagem de entrada; imprima rho e theta da reta; e trace uma reta vermelha. Rode o programa para 2pto.bmp e reta.bmp.

```
> exercicio2 2pto.bmp 2pto.ppm
theta=-74 rho=-20
```



2pto.bmp | 2pto.ppm

```
> exercicio2 reta.bmp reta.ppm
theta=-58 rho=-3
```



reta.bmp | reta.ppm

Rotinas para traçar retas:

OpenCV:
```
void line(Mat& img, Point pt1, Point pt2, const Scalar& color, int thickness=1, int lineType=8, int shift=0)
Ex: Mat_<COR> a; line(a, Point(c1,l1), Point(c2,l2), Scalar(0,0,255));
```

Cekeikon:
```
void reta(Mat_<COR>& b, int l1, int c1, int l2, int c2, COR cor=COR(0,0,0), int largura=1)
Ex: Mat_<COR> a; reta(a, l1, c2, l2, c2, COR(0,0,255));
```

Conversão de sistema (x,y) com centralização para (l,c) com centro no canto superior esquerdo.

```
ImgXyb<FLT> a; a.centro(100,100);
...
int l,c;
int x=10; int y=20;
a.xy2at(x,y,l,c); // converte XY centralizado para LC sem centralizacao
```

Para traçar reta no sistema XY com centro, copie a função abaixo:
```
void retaXyb(ImgXyb<COR>& b, int x1, int y1, int x2, int y2,
             COR cor=COR(0,0,0), int largura=1) {
  int l1,c1,l2,c2;
  b.xy2at(x1,y1,l1,c1);
  b.xy2at(x2,y2,l2,c2);
  reta(b,l1,c1,l2,c2,COR(0,0,255),largura);
}
```

Para converter ImgXyb<FLT> para ImgXyb<COR>:
```
ImgXyb<FLT> a; ...
ImgXyb<COR> b; converte(a,b); b.centro(a.lc,a.cc);
```
A função "converte" converte a matriz, mas não copia o centro.

Sugestão para (PSI3472-2019 Aula 3 Exercício 2 - sugestão): Escreva as funções:

```
void argMin(ImgXyb<FLT> a, int& mintheta, int& minrho);
//Calcula theta e rho com menores valores na imagem do espaço de Hough a

void retaXyb(ImgXyb<COR>& b, int theta, int rho) {
//Traça uma reta theta-rho na imagem colorida b (coordenada xy com centro).
```

(PSI3472-2019 Aula 3 Exercício opcional se der tempo) O mesmo do exercício anterior usando a transformada HoughLines do OpenCV. Note que não há como chamar esta função para devolver somente a reta com o maior de acúmulo no espaço de Hough.

C++ permite deixar a linguagem "do jeito que o usuário quiser". Para quem tem interesse em conhecer detalhe de implementação em C++ da classe ImgXyb. Aqui estou chamando essa classe de "Meu". É o mesmo programa da página anterior.

```
//houghsimp4.cpp - grad2017
#include <cekeikon.h>

template<typename T> class Meu: public Mat_<T> {
public:
  using Mat_<T>::Mat_; //inherit constructors

  T backg;
  int& nx=this->cols;
  int& ny=this->rows;
  int lc=0, cc=0;
  int minx=0, maxx=nx-1, miny=1-ny, maxy=0;

  void centro(int _lc=INT_MIN, int _cc=INT_MIN) {
  // centro() - origem do sistema no centro da imagem
  // centro(0,0) - origem do sistema no canto superior esquerdo
    if (_lc==INT_MIN) {
      lc=(ny-1)/2; cc=(nx-1)/2;
    } else {
      lc=_lc; cc=_cc;
    }
    minx=-cc; maxx=nx-cc-1;
    miny=-(ny-lc-1); maxy=lc;
  }

  Meu<T>() : Mat_<T>() {}

  //<<< copy ctor
  Meu<T>(const Meu<T>& a) : Mat_<T>(a.rows,a.cols) {
    a.copyTo(*this);
    backg=a.backg; lc=a.lc; cc=a.cc;
    minx=a.minx; maxx=a.maxx; miny=a.miny; maxy=a.maxy;
  }

  //<<< copy assign
  Meu<T>& operator=(const Meu<T>& a) {
    if (this == &a) {
      return *this; // beware of self-assignment: x=x
    }
    a.copyTo(*this);
    backg=a.backg; lc=a.lc; cc=a.cc;
    minx=a.minx; maxx=a.maxx; miny=a.miny; maxy=a.maxy;
    return *this;
  }

  //<<< move ctor
  Meu<T>(Meu<T>&& a) {
    *this = a;
    backg=a.backg; lc=a.lc; cc=a.cc;
    minx=a.minx; maxx=a.maxx; miny=a.miny; maxy=a.maxy;
    a.release();
  }

  //<<<move assign
  Meu<T>& operator=(Meu<T>&& a) {
    if (this == &a) return *this; // beware of self-assignment: x=x
    *this = a;
    backg=a.backg; lc=a.lc; cc=a.cc;
    minx=a.minx; maxx=a.maxx; miny=a.miny; maxy=a.maxy;
    a.release();
    return *this;
  }

  //<<< redefinicao do operador (x,y)
  T& operator()(int x, int y) { // modo XY centralizado
    unsigned li=lc-y; unsigned ci=cc+x;
    if (li<unsigned(ny) && ci<unsigned(nx))
      return (static_cast< Mat_<T> >(*this))(li,ci);
```

```
    else return backg;
  }
};

int main() {
  Meu<GRY> ent; le(ent,"2pto.bmp");
  ent.centro(ent.rows/2,ent.cols/2);

  int nl2=teto( sqrt(50*50+50*50) );
  int nl=2*nl2+1;
  int nc2=90;
  int nc=2*nc2+1;

  Meu<FLT> sai(nl,nc,1.0);
  sai.centro(nl2,nc2);

  for (int xa=ent.minx; xa<=ent.maxx; xa++) {
    for (int ya=ent.miny; ya<=ent.maxy; ya++) {
      if (ent(xa,ya)==0) {
        for (int theta=-90; theta<=+90; theta++) {
          double rad=deg2rad(theta);
          int rho=cvRound((xa*cos(rad)+ya*sin(rad)));
          sai(theta,rho) -= 1;
        }
      }
    }
  }
  sai=normaliza(sai);
  imp(sai,"2pto-ho.png");
}
```

Programa completo para detectar retas em imagens binárias de qualquer tamanho.

```
//houghb.cpp
#include <cekeikon.h>

int main(int argc, char** argv)
{ if (argc!=3 && argc!=4 && argc!=5) {
    printf("HoughB: Transformada de Hough de imagem binaria\n");
    printf("HoughB ent.bmp sai.pgm [graus_theta/pix_x_sai] [pix_ent/pix_y_sai]\n");
    printf("Eixo horizontal de sai.tga = theta indo de -90 a +90 graus\n");
    printf("  Se graus_theta/pix_x_sai=2.0, sai tera 91 colunas.\n");
    printf("  Graus_theta/pix_x_sai=1.0 por default\n");
    printf("Eixo vertical de sai.tga = rho com zero central\n");
    printf("  Se pix_ent/pix_y_sai=2.0, cada linha de sai = 2 pixels de ent\n");
    printf("  pix_ent/pix_y_sai=1.0 por default\n");
    printf("Nota: ent.lc=ent.rows/2; ent.cc=ent.cols/2\n");
    erro("Erro: Numero de parametros invalido");
  }

  double gtpxs=1.0;
  if (argc>=4) {
    if (sscanf(argv[3],"%lf",&gtpxs)!=1)
      erro("Erro: leitura graus_theta/pix_x_sai");
  }
  double pepys=1.0;
  if (argc==5) {
    if (sscanf(argv[4],"%lf",&pepys)!=1)
      erro("Erro: leitura pix_ent/pix_y_sai");
  }

  ImgXyb<GRY> ent; le(ent,argv[1]);
  ent.centro(ent.rows/2,ent.cols/2); //Constructor faz diferente

  int nl2=teto( sqrt(double( elev2(ent.lc)+elev2(ent.cc)))/pepys );
  int nl=2*nl2+1;
  // indexacao de rho vai de -nl2 ent +nl2.
  // y=rho/pepys;

  int nc2=teto(90.0/gtpxs);
  int nc=2*nc2+1;
  // indexacao de theta vai de -nc2 ent +nc2
  // x=theta/gtpxs;

  ImgXyb<FLT> sai(nl,nc,0.0);
  sai.centro(nl2,nc2);

  for (int xa=ent.minx; xa<=ent.maxx; xa++) {
    for (int ya=ent.miny; ya<=ent.maxy; ya++)
      if (ent(xa,ya)==0) {
        for (int theta=-nc2; theta<=nc2; theta++) {
          double rad=gtpxs*(M_PI*double(theta)/180);
          int rho=arredonda((xa*cos(rad)+ya*sin(rad))/pepys);
          sai(theta,rho)-=0.01;
        }
      }
  }
  sai=normaliza(sai);
  imp(sai,argv[2]);
}
```

O seguinte programa de lote (batch) em Proeikon detecta segmentos na imagem fig1.jpg:

```
img threshg fig1.jpg f2.bmp 130
img findedb f2.bmp f3.bmp
img houghb f3.bmp hou.tga 0.25
img threshg hou.tga hou2.bmp 160
img erosaob hou2.bmp hou3.bmp 7 7
img findceh hou3.bmp hou4.bmp
img hou2tr hou4.bmp hou5.txt 0.25
img marcatr fig1.jpg hou5.txt hou6.tga
img inicfim f3.bmp hou5.txt hou6.txt 40 2
img marcreta fig1.jpg hou6.txt hou7.tga
```

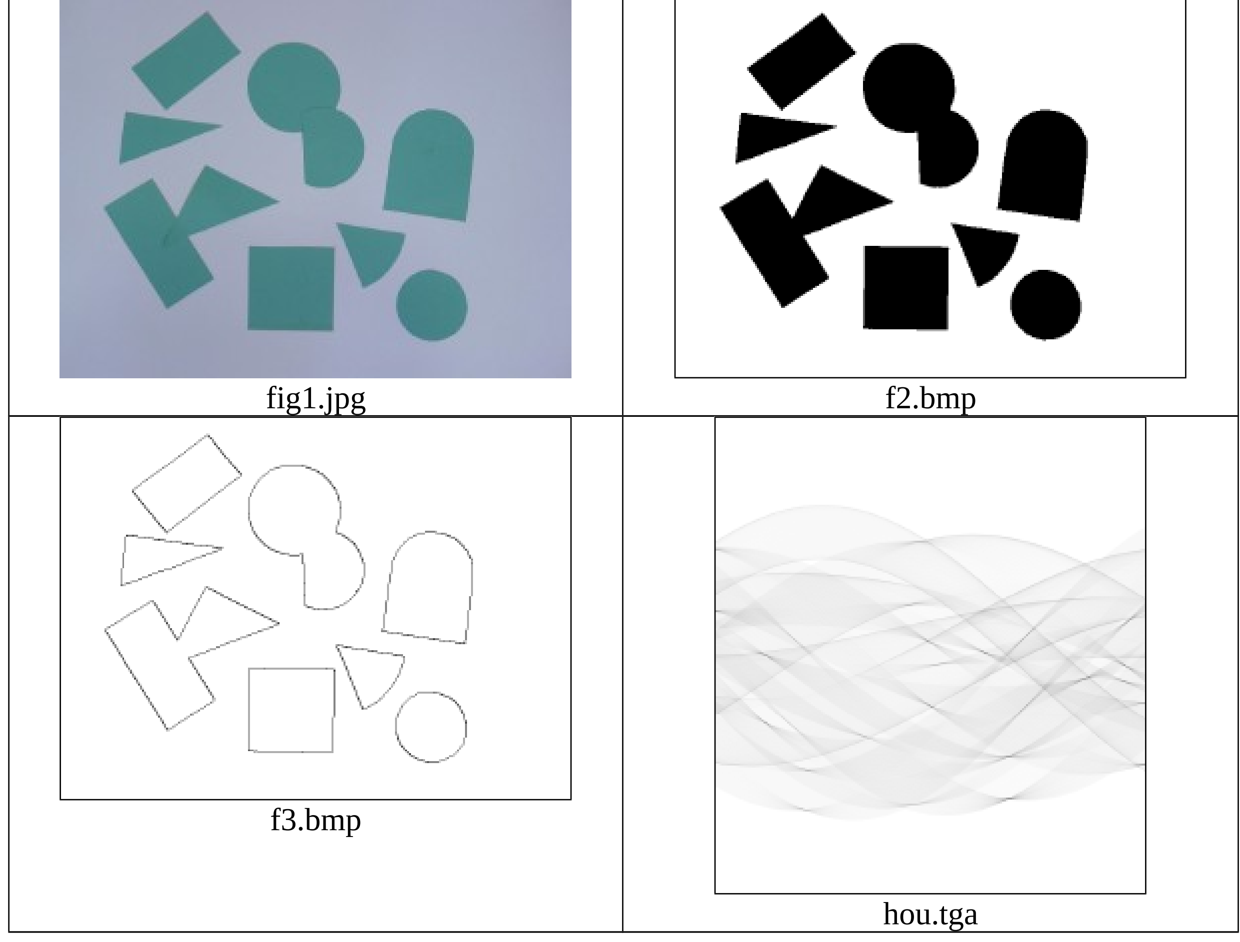
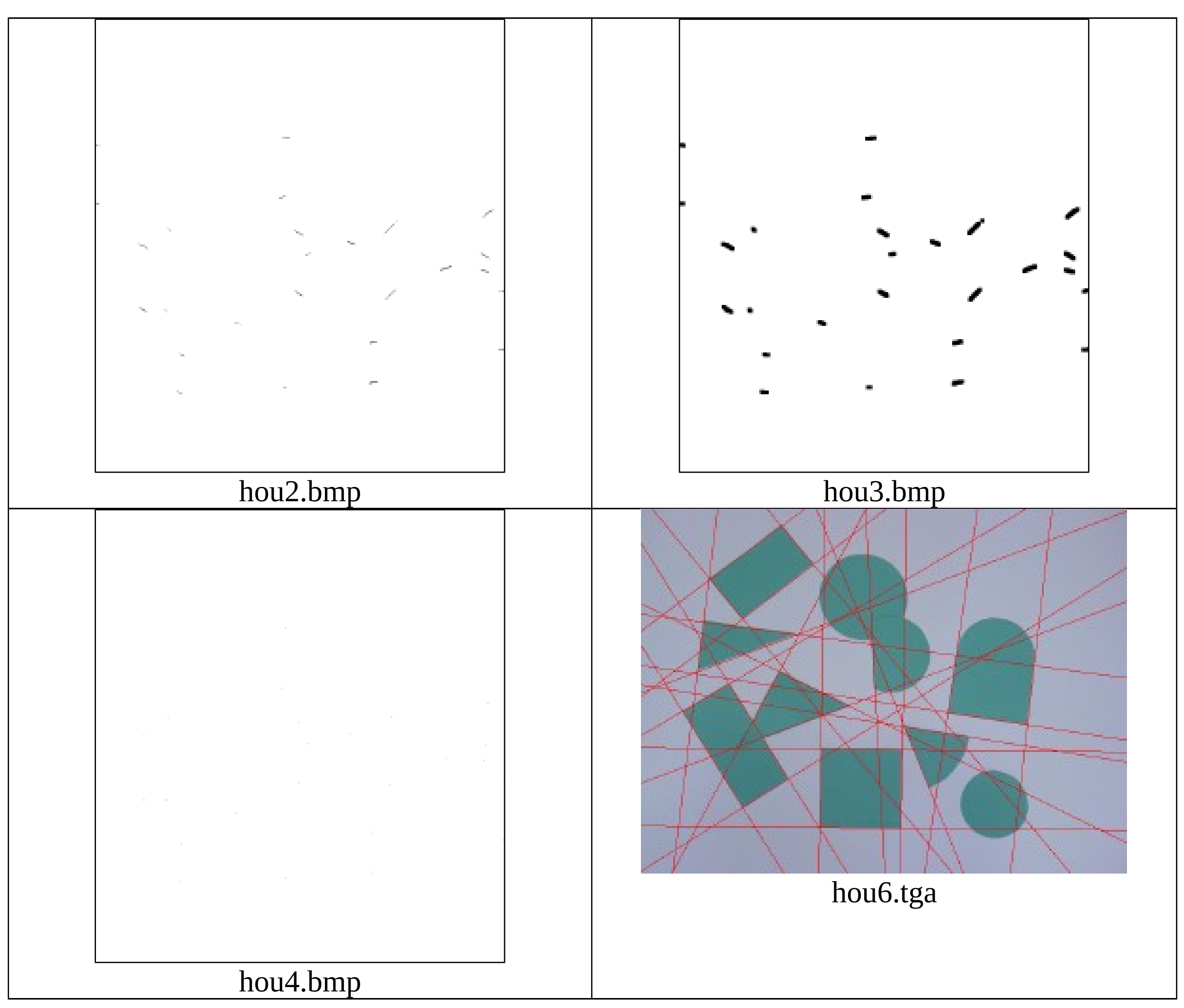
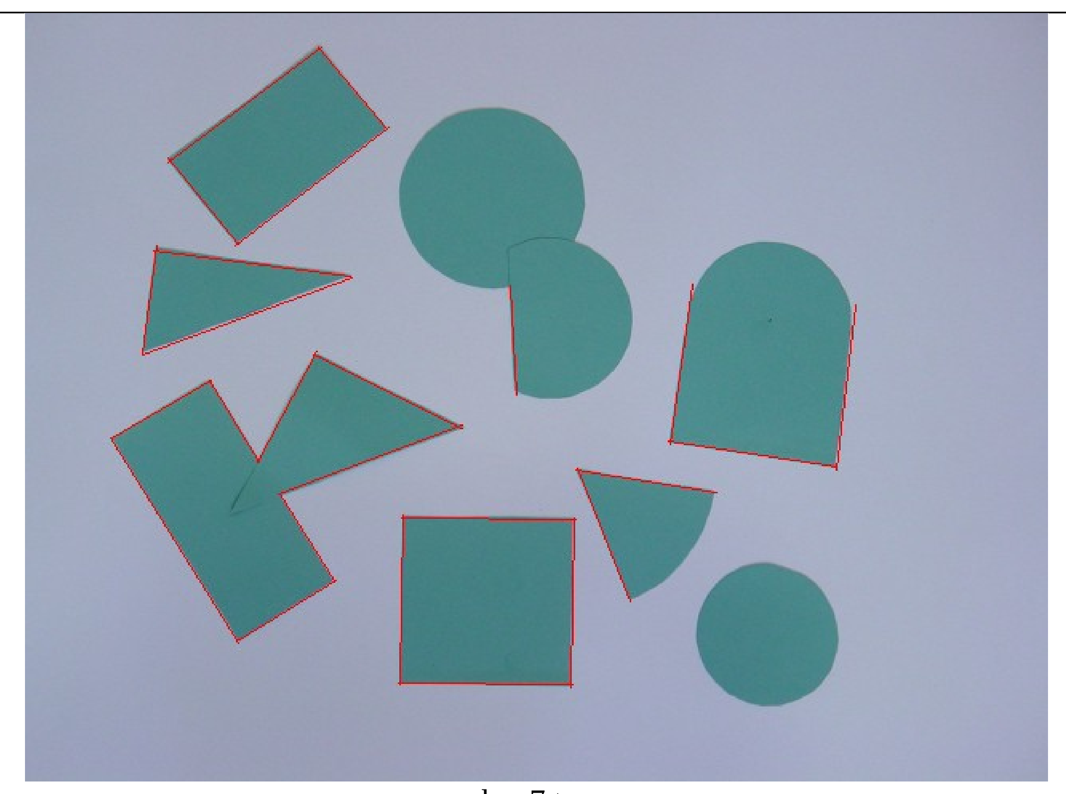


fig1.jpg	f2.bmp
f3.bmp	hou.tga



hou2.bmp hou3.bmp hou4.bmp hou6.tga



hou7.tga

```
24 2
-69.5 -110
-69.5 1
-59.5 -112
-58 31
-53.25 -255
-52.5 -189
-28 -134
-8.25 87
-7 -247
-6.5 192
-1 -82
-1 25
3 -12
22 7
32 -239
32 -168
39.5 -83
39.75 35
63.75 -38
81 -42
81.25 -15
82.5 59
89.5 -180
89.5 -77

hou5.txt
```

```
25 4
213 73 165 204
300 158 257 273
265 54 229 116
392 132 354 194
93 89 21 185
145 131 71 227
280 145 211 182
269 403 169 417
212 73 145 82
285 507 182 519
419 234 313 236
421 341 315 343
238 307 170 303
367 378 284 345
393 133 265 53
354 193 300 159
280 146 229 115
144 133 90 89
72 226 21 183
259 273 213 180
299 432 285 344
283 507 267 402
164 203 148 80
419 342 418 233
316 344 315 235

hou6.txt
```

Aplicação: Detectar e corrigir rotação do documento escaneado (Proeikon).

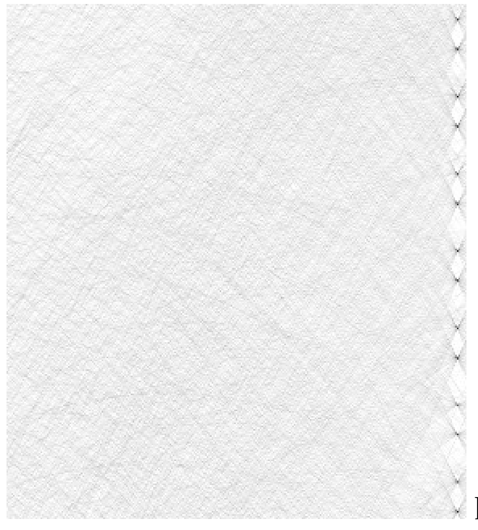
```
::roda.bat
img uplowb rt.bmp uplow.bmp
img houghb uplow.bmp hough.tga 0.125 1
img threshg hough.tga hough.bmp 110
img findceh hough.bmp ce.bmp
img hou2tr ce.bmp retas.mat 0.125 1
img marcatr rt.bmp retas.mat retas.jpg98
```

Aplicação: Corrigir rotação do documento escaneado.

implicariam alto conteúdo de água, não levado em conta no modelo. Por interferência de John Randall (King's College) com Bragg, Watson e Crick tiveram de desistir de trabalhar com DNA, deixando isso para o pessoal do King's College. Crick voltou para sua tese

rt.bmp

uplow.bmp


hough.tga

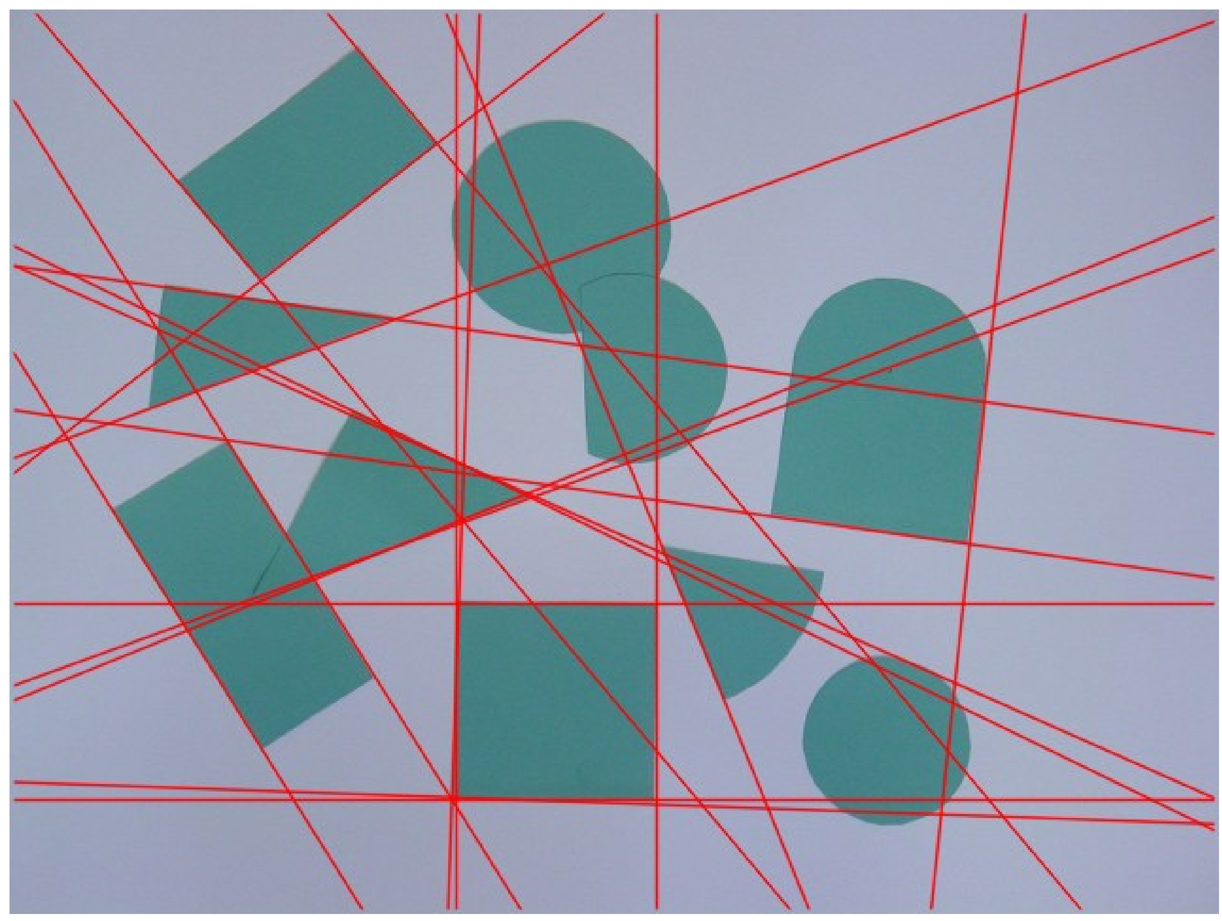
implicariam alto conteúdo de água, não levado em conta no modelo. Por interferência de John Randall (King's College) com Bragg, Watson e Crick tiveram de desistir de trabalhar com DNA, deixando isso para o pessoal do King's College. Crick voltou para sua tese com a hemoglobina e Watson induziu

retas.jpg98

A transformada de Hough para detectar retas do OpenCV não funciona bem.

//HoughCV.cpp pos2014
```
#include <cekeikon.h>

int main()
{ Mat_<GRY> a; le(a,"f3.bmp");
  { using namespace Morphology;
    a=-a;   // imagem deve ser backg preto e foreg branco
  }
  vector<Vec2f> lines;
  HoughLines(a, lines, 1, deg2rad(2), 60);
  // void HoughLines(InputArray image, OutputArray lines,
  //                  double rho, double theta, int threshold,
  //                  double srn=0, double stn=0 )
  for (unsigned i=0; i<lines.size(); i++) {
    printf("%u %g %g\n",i,lines[i][0],rad2deg(lines[i][1]));
  }

  Mat_<COR> d; le(d,"fig1.jpg");
  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( d, pt1, pt2, Scalar(0,0,255), 1, CV_AA);
  }
  mostra(d);
}
```

A função HoughLines retorna somente uma lista de coordenadas $(\rho;\theta)$ onde se acumulou mais de um certo número de pontos. Não retorna a quantidade acumulada para cada uma entrada $(\rho;\theta)$ da lista. Assim, não dá para detectar (por exemplo) a reta com o maior acúmulo no espaço de Hough. Outro problema (que veremos adiante) é que deve-se detectar as arestas como imagem binária antes de chamar esta função. A transformada de Hough mais robusta e eficiente é usar gradiente e sem detectar as arestas previamente.



Saída gerada pela função HoughLines do OpenCV.

Uso de gradiente para acelerar transformada de Hough:

Parece que não há implementação de OpenCV que utilize gradiente para acelerar transformada de Hough para detectar retas.

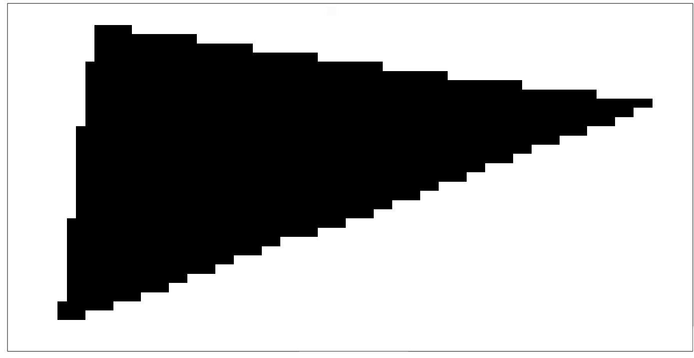
Programa abaixo detecta gradiente de uma imagem em níveis de cinza e o grava como uma imagem complexa.

```
//<<<<<<<<< Programa GradienG (incluído na KCEK > 5.26) <<<<<<<<<

//Funciona para Cekeikon > 5.26
//gradieng.cpp
#include <cekeikon.h>

int main(int argc, char** argv) {
  if (argc!=3 && argc!=4) {
    printf("GradienG: Calcula gradiente de Mat_<GRY> como Mat_<CPX>\n");
    printf("GradienG ent.pgm sai.img [gaussDev]\n");
    xerro1("Erro: Numero de argumentos invalido");
  }
  Mat_<GRY> a;
  le(a,argv[1]);
  if (argc==4) {
    double gaussDev;
    convArg(gaussDev,argv[3]);
    GaussianBlur(a,a,Size(0,0),gaussDev,gaussDev);
  }
  Mat_<CPX> b=gradienteScharr(a,true); // true=y para cima
  imp(b,argv[2]);
}

//Nao copie daqui para baixo,
//pois estas funcoes ja estao incluidas na biblioteca Cekeikon
//<<<<<<<<< Função gradienteScharr (incluído na Cekeikon > 5.26) <<<<<<<<<

Mat_<CPX> criaCPX(Mat_<FLT> dc, Mat_<FLT> dl)
{ if (dc.size()!=dl.size()) erro("Erro criaCPX");
  Mat_<CPX> b(dc.size());
  for (unsigned i=0; i<b.total(); i++)
    b(i)=CPX( dc(i), dl(i) );
  return b;
}

Mat_<CPX> gradienteScharr(Mat_<GRY> a, bool yParaCima)
// x ->
// y |
//   v
{ Mat_<FLT> gradientex;
  Scharr(a, gradientex, CV_32F, 1, 0);
  // gradientex entre -4080 e +4080 (255*16)?
  gradientex=(1.0/4080)*gradientex;

  Mat_<FLT> gradientey;
  Scharr(a, gradientey, CV_32F, 0, 1);
  // bl entre -4080 e +4080?
  if (yParaCima) gradientey=(-1.0/4080)*gradientey;
  else gradientey=(1.0/4080)*gradientey;

  Mat_<CPX> b=criaCPX(gradientex,gradientey);
  return b; // b.real entre -1 e +1. b.imag tambem
}
```

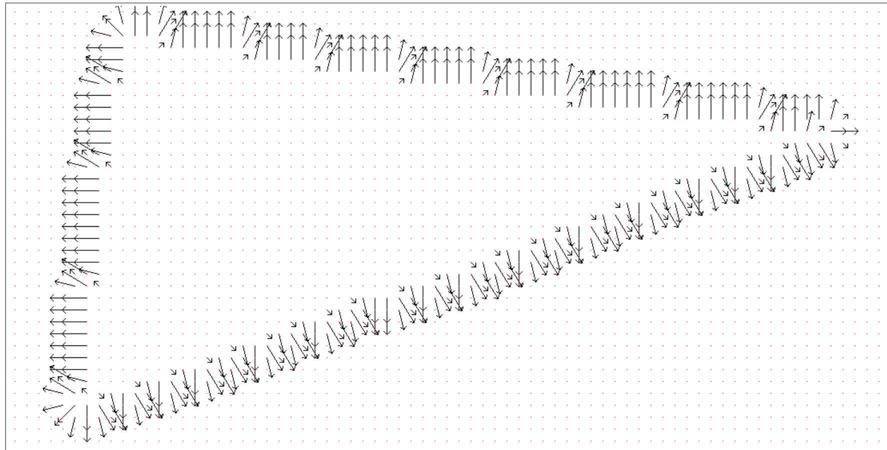
```
Nota: CPX é equivalente a complex<float>

Pegar parte real: a(l,c).real()  ou  real(a(l,c))
Pegar parte imaginária: a(l,c).imag()  ou  imag(a(l,c))
Módulo: abs(a(l,c))
Ângulo em radianos: arg(a(l,c))
Atribuir um número complexo: a(l,c)=CPX(0,1)
```

```
#!/bin/bash
#roda.sh
kcek gradieng triang.png triang0.img
kcek campox triang0.img triang0.png 31 15
kcek gradieng triang.png triang2.img 2
kcek campox triang2.img triang2.png 31 15
kcek mostrax triang2.img
```

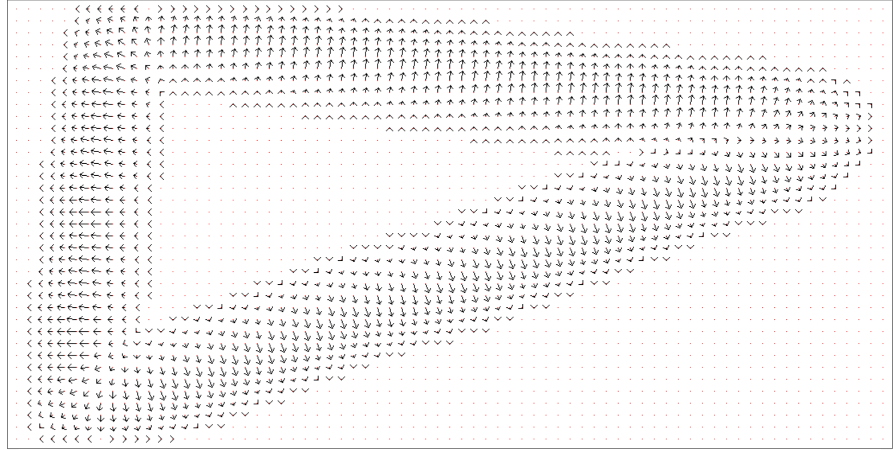
Nota: Se pedir para gravar uma imagem com nome "win", Cekeikon mostra a imagem na tela.



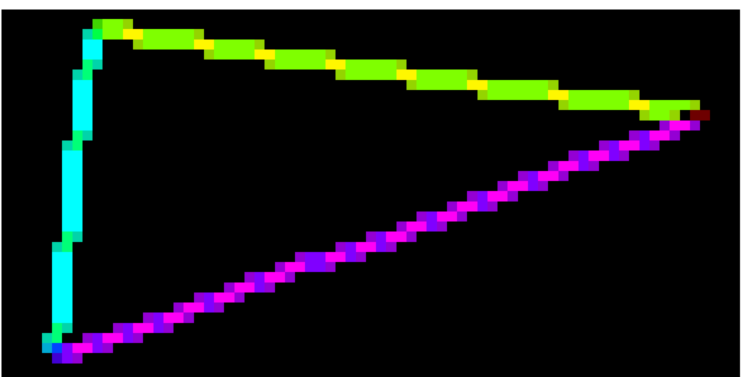
triang.png



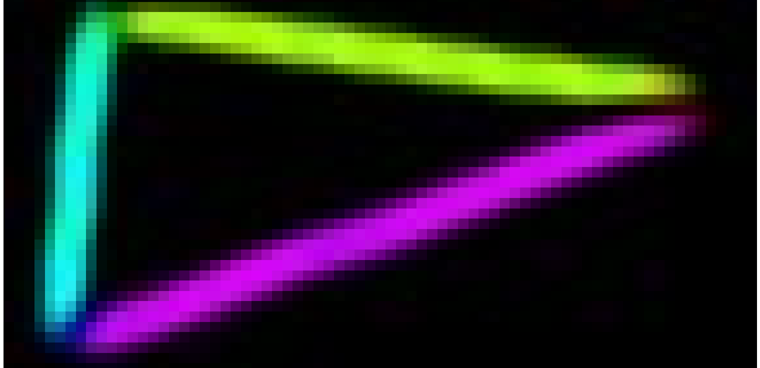
triang0.png. Sem filtragem gaussiana. Alguns vetores têm direção errada.



triang2.png. Filtragem gaussiana sigma=desvio=2 pixels. Menos erro na direção.



triang0.img (sem filtro gaussiano) usando brilho como magnitude e cor como direção do gradiente. A direção do gradiente não é constante numa aresta.



triang2.img (com filtro gaussiano de sigma=2 pixels) usando brilho como magnitude e cor como direção do gradiente. A direção do gradiente é mais ou menos constante numa aresta.

(PSI3472-2019 Aula 3 Exercício 3) Rode gradieng.cpp para triang.png, para obter gradientes desenhados como flechas e como imagem em sistema HSI, sem e com filtro gaussiano de desvio-padrão de 2 pixels.

Calcula transformada de Hough para detectar retas, usando módulo de gradiente (sem binarizar imagem). O programa abaixo é simplificado (por motivos didáticos) e só funciona para imagem de entrada em níveis de cinza 100x100. Subtrai o módulo do gradiente no espaço de Hough.

```
//houghgrsimp1.cpp
#include <cekeikon.h>

Mat_<CPX> anguloMaisMenosPi2(Mat_<CPX> a) {
  Mat_<CPX> d(a.size());
  for (unsigned i=0; i<a.total(); i++) {
    float rad=a(i).imag();
    if (rad>M_PI) rad -= M_PI;
    if (rad<=-M_PI) rad += M_PI;
    assert(-M_PI<rad && rad<=M_PI);
    if (rad<0) rad+=M_2PI;
    assert(0<=rad && rad<=2*M_PI);
    if (rad>M_PI) rad-=M_PI;
    assert(0<=rad && rad<=M_PI);
    if (rad>M_PI_2) rad-=M_PI;
    assert(-M_PI_2<=rad && rad<=M_PI_2);
    // rad entre -M_PI_2 e +M_PI_2
    d(i)=CPX( a(i).real(), rad );
  }
  return d;
}

int main() {
  ImgXyb<GRY> ent; le(ent,"trianghd.png");
  ent.centro(ent.rows/2,ent.cols/2);

  ImgXyb<CPX> ret=gradienteScharr(ent,true); // y para cima
  ret.centro(ret.rows/2,ret.cols/2); //Constructor faz diferente
  ImgXyb<CPX> pol=ret2pol(ret); // pol com modulo e angulo em radianos
                                // -pi<arg<=+pi
  pol=anguloMaisMenosPi2(pol); // -pi/2<arg<=+pi/2
  pol.centro(pol.rows/2,pol.cols/2);

  int nl2=teto( sqrt(50*50+50*50) );
  int nl=2*nl2+1;
  int nc2=90;
  int nc=2*nc2+1;

  ImgXyb<FLT> sai(nl,nc,0.0);
  sai.centro(nl2,nc2);

  float limiar=0.1;
  for (int xa=pol.minx; xa<=pol.maxx; xa++) {
    for (int ya=pol.miny; ya<=pol.maxy; ya++) {
      if (pol(xa,ya).real()>=limiar) {
        for (int theta=-90; theta<=+90; theta++) {
          double rad=deg2rad(theta);
          int rho=round((xa*cos(rad)+ya*sin(rad)));
          sai(theta,rho) -= pol(xa,ya).real();
        }
      }
    }
  }
  sai=normaliza(sai);
  imp(sai,"trianghd-hough.png");
}
```

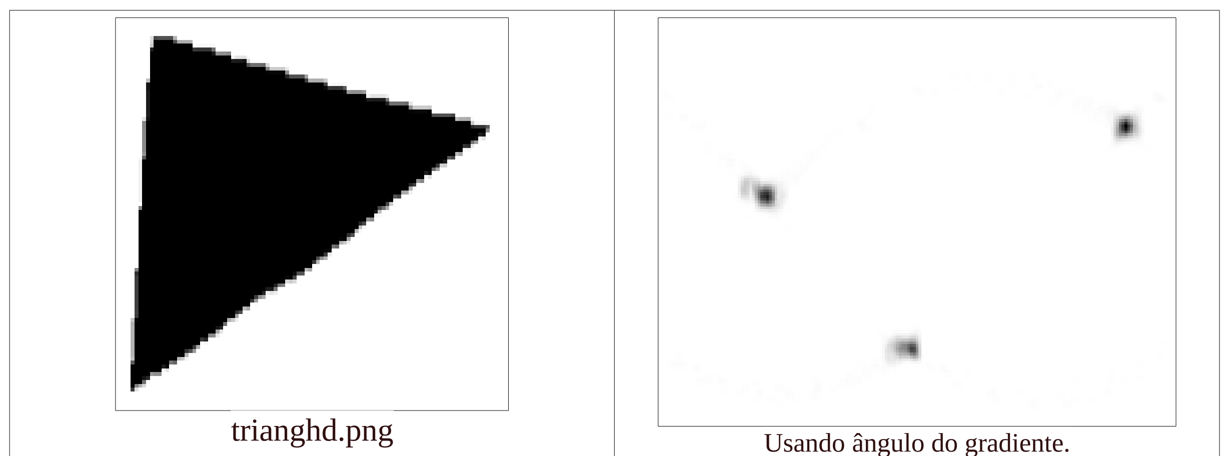
trianghd.png	Sem usar ângulo do gradiente.

Calcula transformada de Hough para detectar retas, usando módulo e ângulo de gradiente (sem binarizar imagem). O programa abaixo é simplificado (por motivos didáticos) e só funciona para imagem de entrada em níveis de cinza 100x100.

```
//houghgrsimp5.cpp - pos2017
#include <cekeikon.h>

Mat_<CPX> anguloMaisMenosPi2(Mat_<CPX> a) {
  Mat_<CPX> d(a.size());
  for (unsigned i=0; i<a.total(); i++) {
    float rad=a(i).imag();
    if (rad>M_PI) rad -= M_PI;
    if (rad<=-M_PI) rad += M_PI;
    assert(-M_PI<rad && rad<=M_PI);

    if (rad<0) rad+=M_2PI;
    assert(0<=rad && rad<=2*M_PI);

    if (rad>M_PI) rad-=M_PI;
    assert(0<=rad && rad<=M_PI);

    if (rad>M_PI_2) rad-=M_PI;
    assert(-M_PI_2<=rad && rad<=M_PI_2);
    // rad entre -pi/2 e +pi/2

    d(i)=CPX( a(i).real(), rad );
  }
  return d;
}

void subtrai(ImgXyb<FLT>& sai, int x, int y, FLT val, int dx, int dy) {
  for (int x2=-dx; x2<=dx; x2++)
    for (int y2=-dy; y2<=dy; y2++)
      sai(x+x2,y+y2) -= val;
}

int main() {
  ImgXyb<GRY> ent; le(ent,"trianghd.png");
  ent.centro(ent.rows/2,ent.cols/2);
  GaussianBlur(ent,ent,Size(0,0),2.0,2.0);

  ImgXyb<CPX> ret=gradienteScharr(ent,true); // y para cima
  ret.centro(ret.rows/2,ret.cols/2); //Constructor faz diferente
  ImgXyb<CPX> pol=ret2pol(ret); // pol com modulo e angulo em radianos
                                // -pi<arg<=+pi
  pol=anguloMaisMenosPi2(pol); // -pi/2<arg<=+pi/2
  pol.centro(pol.rows/2,pol.cols/2);

  int nl2=teto( sqrt(50*50+50*50) );
  int nl=2*nl2+1;
  int nc2=90;
  int nc=2*nc2+1;

  ImgXyb<FLT> sai(nl,nc,0.0);
  sai.centro(nl2,nc2);

  float limiar=0.1;
  // limiar pode ate ser zero que o processamento e' rapido.
  for (int xa=pol.minx; xa<=pol.maxx; xa++) {
    for (int ya=pol.miny; ya<=pol.maxy; ya++) {
      if (pol(xa,ya).real()>=limiar) {
        double rad=pol(xa,ya).imag();
        int rho=round((xa*cos(rad)+ya*sin(rad)));
        int theta=chao(rad2deg(rad));
        assert(-90<=theta && theta<=90);
        subtrai(sai,theta,rho,pol(xa,ya).real(),1,1);
        //sai(theta,rho) -= pol(xa,ya).real();
      }
    }
  }
  sai=normaliza(sai);
  imp(sai,"trianghd-hough5.png");
}
```

Pergunta: Por que é lícito converter o ângulo do gradiente, de -180 a +180 graus para -90 a +90 graus? Isto é, por que se pode ignorar o sentido do gradiente?

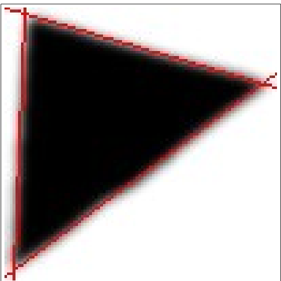
Nota: Há o programa "kcek houghgr" já pronto.



trianghd.png	Usando ângulo do gradiente.

Saída trianghd-hough5.png. Usando ângulo do gradiente.

(PSI3472-2019 Aula 3 Exercício 4) Execute houghgrsimp1.cpp e houghgrsimp5.cpp para detectar as 3 retas em trianghd.png. Modifique o programa houghgrsimp5.cpp para que detecte as 3 retas-suportes na imagem de entrada; imprima rho e theta das 3 retas; e trace uma reta vermelha para cada uma das 3 retas detectadas.



(PSI3472-2019 Aula 3 Exercício opcional se der tempo) O mesmo do exercício anterior usando HoughLines do OpenCV.

HoughLines [do manual do OpenCV2]:
Finds lines in a binary image using the standard Hough transform.

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

Parameters
image – 8-bit, single-channel binary source image. The image may be modified by the function.
lines – Output vector of lines. Each line is represented by a two-element vector $(\rho;\theta)$. ρ is the distance from the coordinate origin (0;0) (top-left corner of the image). θ is the line rotation angle in radians (0 is vertical line; $\pi/2$ is horizontal line).
rho – Distance resolution of the accumulator in pixels.
theta – Angle resolution of the accumulator in radians.
threshold – Accumulator threshold parameter. Only those lines are returned that get enough votes (> threshold).
srn – For the multi-scale Hough transform, it is a divisor for the distance resolution rho. The coarse accumulator distance resolution is rho and the accurate accumulator resolution is rho/srn . If both srn=0 and stn=0 , the classical Hough transform is used. Otherwise, both these parameters should be positive.
stn – For the multi-scale Hough transform, it is a divisor for the distance resolution theta.

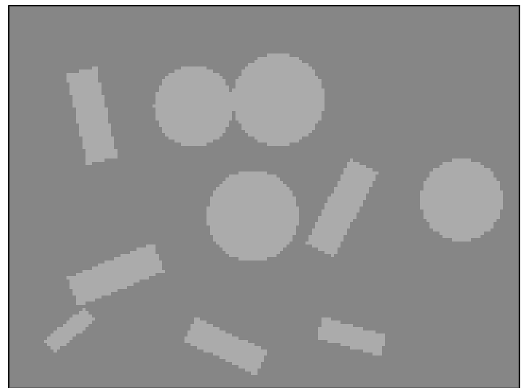
method – One of the following Hough transform variants:
– CV_HOUGH_STANDARD classical or standard Hough transform. Every line is represented by two floating-point numbers ($\rho;\theta$), where ρ is a distance between (0,0) point and the line, and θ is the angle between x-axis and the normal to the line. Thus, the matrix must be (the created sequence will be) of CV_32FC2 type.
– CV_HOUGH_PROBABILISTIC probabilistic Hough transform (more efficient in case if the picture contains a few long linear segments). It returns line segments rather than the whole line. Each segment is represented by starting and ending points, and the matrix must be (the created sequence will be) of the CV_32SC4 type.
– CV_HOUGH_MULTI_SCALE multi-scale variant of the classical Hough transform.
The lines are encoded the same way as CV_HOUGH_STANDARD.
param1 – First method-dependent parameter:
– For the classical Hough transform, it is not used (0).
– For the probabilistic Hough transform, it is the minimum line length.
– For the multi-scale Hough transform, it is srn.
param2 – Second method-dependent parameter:
– For the classical Hough transform, it is not used (0).
– For the probabilistic Hough transform, it is the maximum gap between line segments lying on the same line to treat them as a single line segment (that is, to join them).
– For the multi-scale Hough transform, it is stn.
The function implements the standard or standard multi-scale Hough transform algorithm for line detection. See http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm for a good explanation of Hough transform. See also the example in HoughLinesP() description.

Programa completo houghgr.cpp (calcula Hough a partir de gradiente, sem detectar arestas). Está incluída no Cekeikon (programa kcek houghgr).

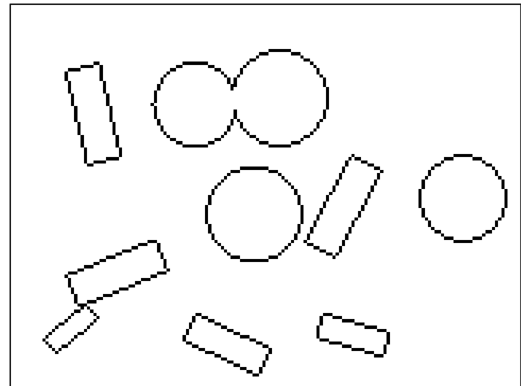
```
//houghgr.cpp
#include <cekeikon.h>

Mat_<CPX> anguloMaisMenosPi2(Mat_<CPX> a) {
  Mat_<CPX> d(a.size());
  for (unsigned i=0; i<a.total(); i++) {
    float rad=a(i).imag();
    if (rad>M_PI) rad -= M_PI;
    if (rad<=-M_PI) rad += M_PI;
    assert(-M_PI<rad && rad<=M_PI);

    if (rad<0) rad+=M_2PI;
    assert(0<=rad && rad<=2*M_PI);

    if (rad>M_PI) rad-=M_PI;
    assert(0<=rad && rad<=M_PI);

    if (rad>M_PI_2) rad-=M_PI;
    assert(-M_PI_2<=rad && rad<=M_PI_2);
    // rad entre -pi/2 e +pi/2

    d(i)=CPX( a(i).real(), rad );
  }
  return d;
}

void subtrai(ImgXyb<FLT>& sai, int x, int y, FLT val, int dx, int dy) {
  for (int x2=-dx; x2<=dx; x2++)
    for (int y2=-dy; y2<=dy; y2++)
      sai(x+x2,y+y2) -= val;
}

ImgXyb<FLT> houghGr(ImgXyb<GRY> ent, double resRho=1.0, double resTheta=1.0,
            double GaussDev=2.0, double minGrad=0.05) {
// Detecta retas usando gradiente da imagem, sem binarizar.
// ent = imagem em niveis de cinza (e nao binaria). Retas sao as arestas da imagem.
// resRho = pix_ent/pix_y_sai = pepys = distance resolution of the accumulator in pixels.
// resTheta = graus_theta/pix_x_sai = gtpxs = angle resolution of the accumulator in degrees.
// GaussDev = sigma ou desvio-padrao do filtro gaussiano (para calcular gradiente).
// minGrad = menor valor do modulo de gradiente para que seja processado.
//           se for zero, todos os pixels sao processados.
//           O maior modulo de gradiente e' 1.
// saida = imagem no espaco de Hough, normalizado entre 0 e 1.
  ent.centro();
  GaussianBlur(ent,ent,Size(0,0),GaussDev,GaussDev);

  ImgXyb<CPX> ret=gradienteScharr(ent,true); // y para cima
  ret.centro();
  ImgXyb<CPX> pol=ret2pol(ret); // pol com modulo e angulo em radianos
                                // -pi<arg<=+pi
  pol=anguloMaisMenosPi2(pol); // -pi/2<arg<=+pi/2
  pol.centro();

  int nl2=teto( sqrt(double( elev2(ent.lc)+elev2(ent.cc)))/resRho );
  int nl=2*nl2+1;
  // indexacao de rho vai de -nl2 ent +nl2.
  // y=rho/resRho;

  int nc2=teto(90.0/resTheta);
  int nc=2*nc2+1;
  // indexacao de theta vai de -nc2 ent +nc2
  // x=theta/resTheta;

  ImgXyb<FLT> sai(nl,nc,0.0);
  sai.centro(nl2,nc2);

  for (int xa=pol.minx; xa<=pol.maxx; xa++) {
    for (int ya=pol.miny; ya<=pol.maxy; ya++) {
      if (pol(xa,ya).real()>=minGrad) {
```

```
        double rad=pol(xa,ya).imag();
        int rho=round((xa*cos(rad)+ya*sin(rad))/resRho);
        assert(-nl2<=rho && rho<=nl2);
        int theta=chao(rad2deg(rad)/resTheta);
        assert(-nc2<=theta && theta<=nc2);
        subtrai(sai,theta,rho,pol(xa,ya).real(),round(resTheta),round(resRho));
        //sai(theta,rho) -= pol(xa,ya).real();
      }
    }
  }
  sai=normaliza(sai);
  return sai;
}

int main(int argc, char** argv) {
  const char* keys =
    " curto| longo    |default| explicacao\n"
    "  -r  | -rho     | 1     | resolucao de distancia em pixels\n"
    "  -t  | -theta   | 1     | resolucao de angulo em graus\n"
    "  -d  | -desvio  | 2     | desvio do filtro Gaussiano\n"
    "  -m  | -mingrad | 0.05  | modulo do gradiente minimo para processar\n";

  if (argc<=1) {
    printf("HoughGr: Detecta retas usando gradiente da imagem, sem binarizar.\n");
    printf("HoughGr ent.pgm sai.img [opcoes]\n");
    printf("%s",keys);
    erro("Erro: Numero de argumentos invalido");
  }

  ArgComando cmd(argc,argv);
  string nomeent=cmd.getCommand(0);
  if (nomeent=="") erro("Erro: Nao especificou imagem de entrada");
  Mat_<GRY> ent; le(ent,nomeent);

  string nomesai=cmd.getCommand(1);
  if (nomesai=="") erro("Erro: Nao especificou imagem de saida");

  double resRho=cmd.getDouble("-r","-rho",1.0);
  double resTheta=cmd.getDouble("-t","-theta",1.0);
  double GaussDev=cmd.getDouble("-d","-desvio",2.0);
  double minGrad=cmd.getDouble("-m","-mingrad",0.05);
  Mat_<FLT> sai=houghGr(ent,resRho,resTheta,GaussDev,minGrad);
  imp(sai,nomesai);
}
```

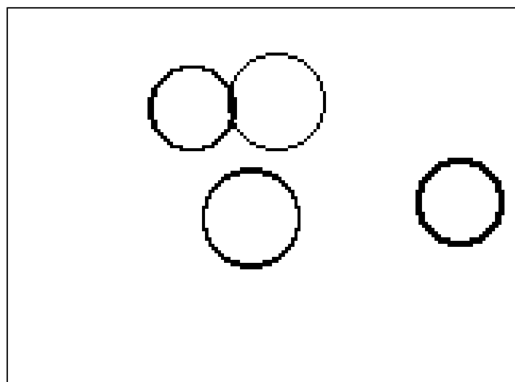
Detecção de círculos pela transformada de Hough:



Original



Arestas

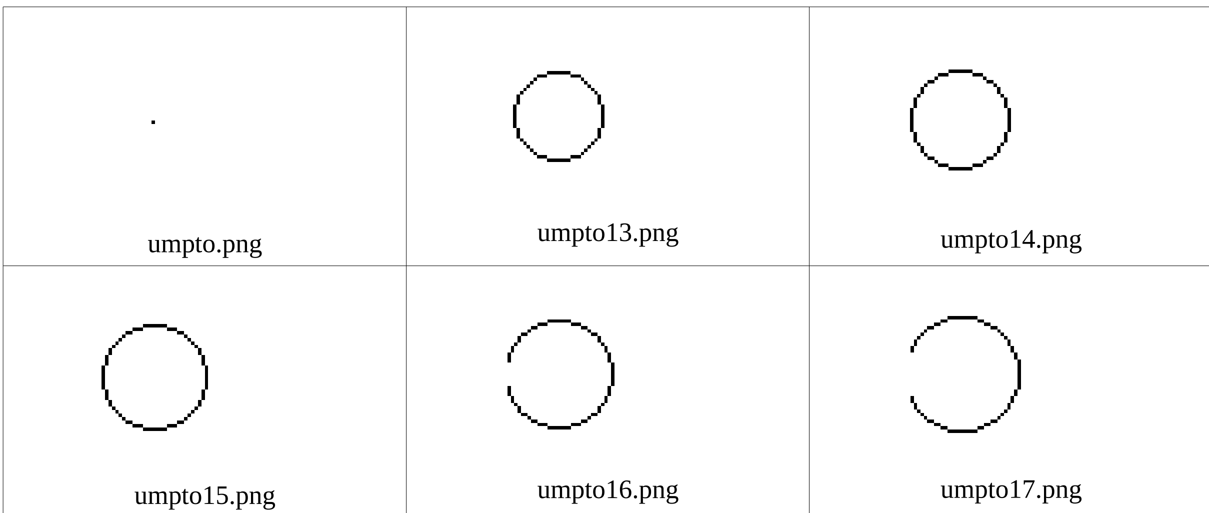


Círculos:

$$r=\sqrt{(l-l_o)^2+(c-c_o)^2}$$

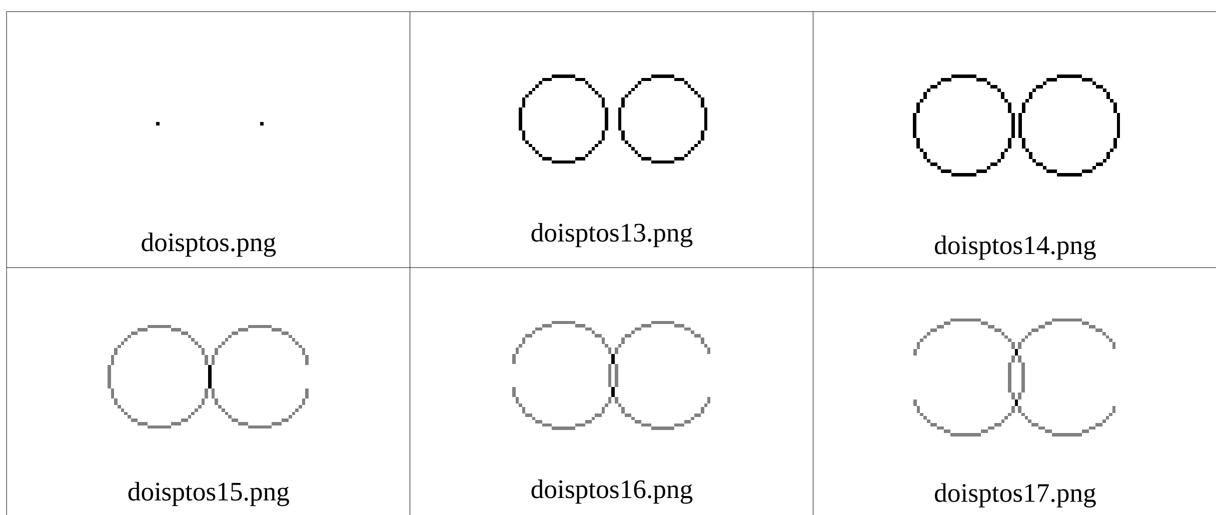
Como funciona Hough para detectar círculos:

A transformada de Hough de um ponto *P* torna-se um círculo em cada fatia do volume 3D do espaço de Hough. O centro do círculo é a coordenada de *P*. Os raios dos círculos são os índices das fatias.



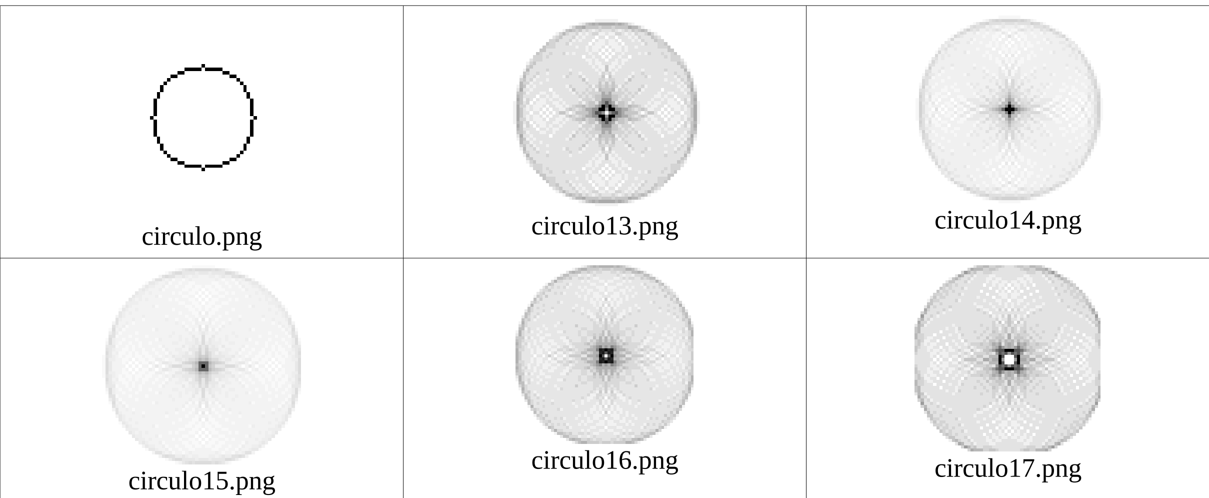

```
>houghcir umpto.png umpto 13 17
>kcek houcirb umpto.png umpto 13 17
```

A transformada de Hough de dois pontos P e Q torna-se dois círculos em cada fatia do volume 3D de Hough. Cada intersecção de dois círculos numa fatia indica o centro de um círculo que passa pelos pontos P e Q.



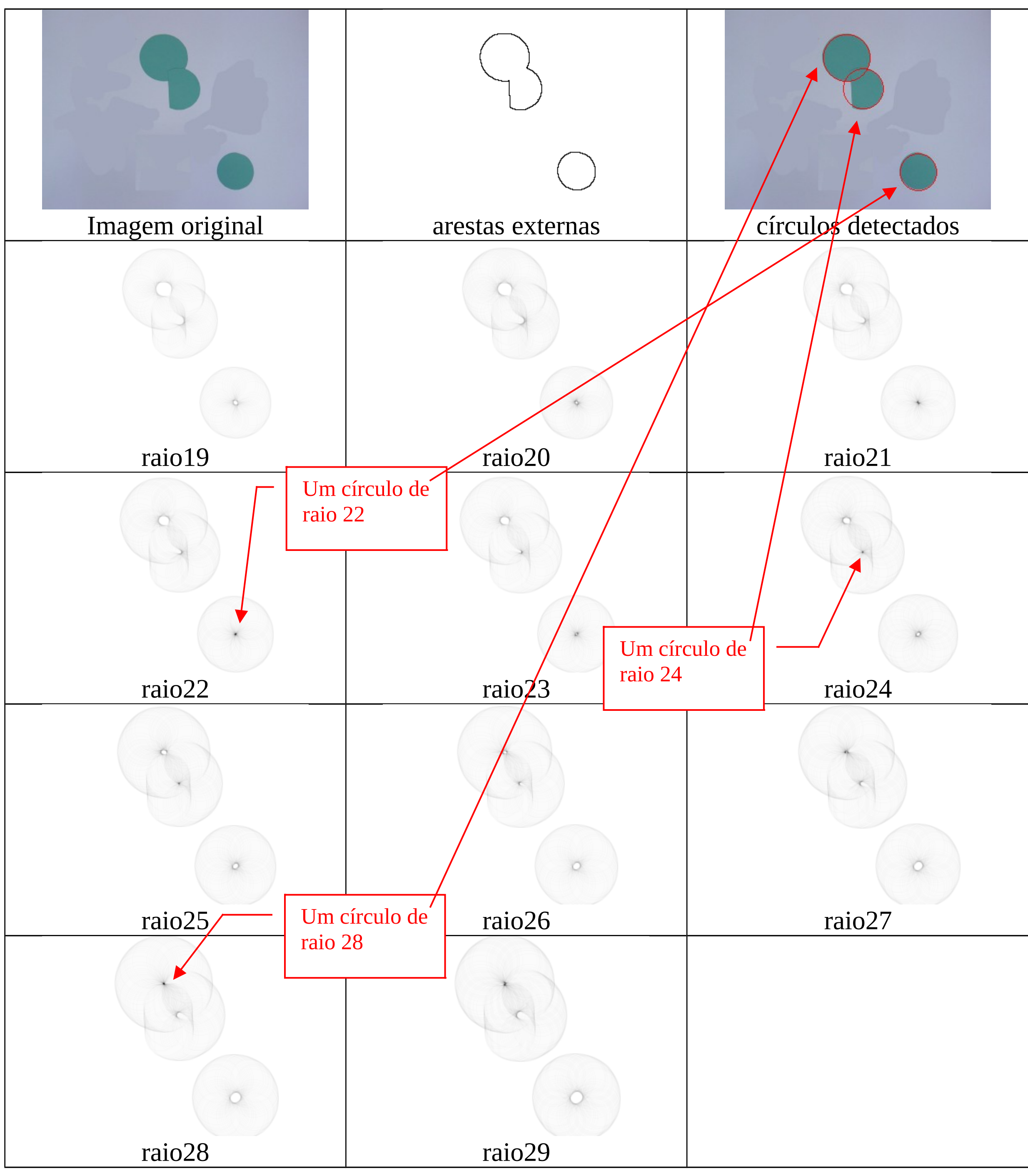

```
>houghcir doisptos.png doisptos 13 17
>kcek houcirb doisptos.png doisptos 13 17
```

A transformada de Hough de um círculo C são um número grande de círculos. O ponto de encontro desses círculos indica o centro de C. O raio é o índice da fatia.




```
>houghcir circulo.png circulo 13 17
>kcek houcirb circulo.png circulo 13 17
```

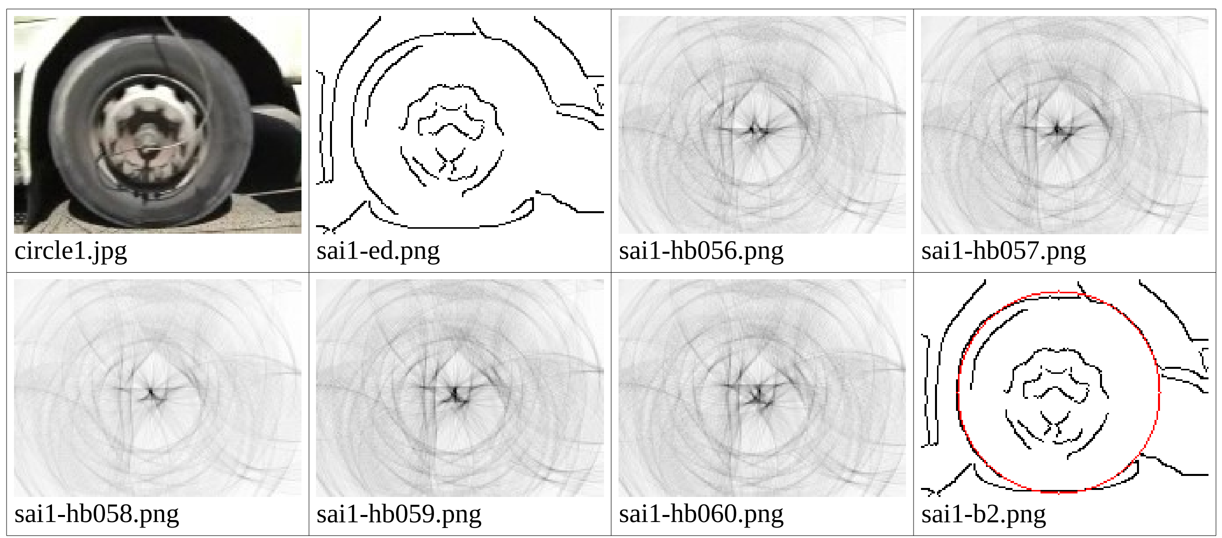
```
img threshg image1.jpg i2.bmp 130
img findedb i2.bmp i3.bmp
img houcir3 i3.bmp 18 32 circ.img circ
```



É possível melhorar consideravelmente a velocidade de processamento utilizando a informação sobre a orientação local das arestas.

Programa HOUCIRB.CPP (2017): (houcirb.cpp)

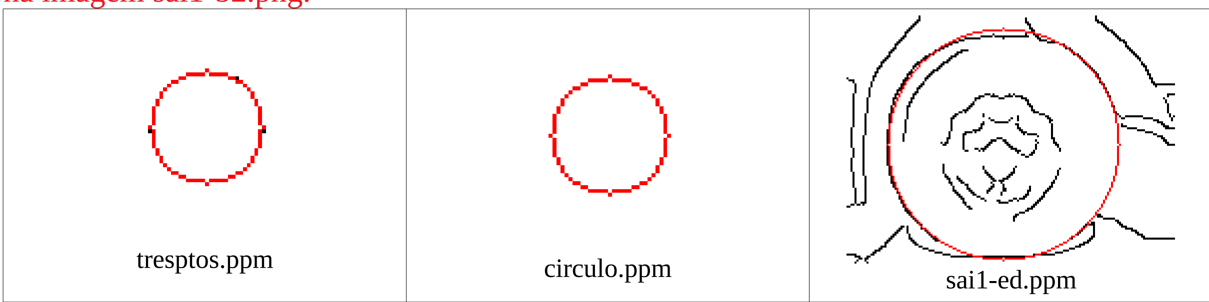
```
//houcirb.cpp - grad2017
#include <cekeikon.h>

void subt(vector< Mat_<FLT> >& b, int s, int l, int c) {
  if (0<=s && s<int(b.size()) && 0<=l && l<b[0].rows && 0<=c && c<b[0].cols)
    b[s](l,c)-=1.0;
}

void subtraicirculo(vector< Mat_<FLT> >& b, int s, int l, int c, int r) {
  int x=0; int y=r; int limite=r;
  while (y>0) {
    subt(b,s,l+x,c+y);
    subt(b,s,l+y,c-x);
    subt(b,s,l-x,c-y);
    subt(b,s,l-y,c+x);
    int mh=abs(elev2(x+1)+elev2(y)-elev2(r));
    int md=abs(elev2(x+1)+elev2(y-1)-elev2(r));
    int mv=abs(elev2(x)+elev2(y-1)-elev2(r));
    if (mh<=md && mh<=mv) x++;
    else if (md<=mv) { x++; y--; }
    else y--;
  }
}

int main(int argc, char** argv)
{ if (argc!=5) {
    printf("HoughCir ent.bmp saida rmin rmax\n");
    printf("  ent.bmp e' imagem binaria, com backg=255 foreg=0.\n");
    erro("Erro: Numero de argumentos invalido");
  }

  int rmin;
  if (sscanf(argv[3],"%d",&rmin)!=1) erro("Erro: Leitura rmin");

  int rmax;
  if (sscanf(argv[4],"%d",&rmax)!=1) erro("Erro: Leitura rmax");

  Mat_<GRY> ent; le(ent,argv[1]);

  int nl=ent.rows;
  int nc=ent.cols;
  int ns=rmax-rmin+1;

  vector< Mat_<FLT> > sai(ns);
  for (int i=0; i<ns; i++) {
    sai[i].create(nl,nc);
    sai[i].setTo(0.0);
  }
  //sai[s](l,c)

  for (int l=0; l<nl; l++)
    for (int c=0; c<nc; c++)
      if (ent(l,c)==0) {
        for (int r=rmin; r<=rmax; r++)
          subtraicirculo(sai,r-rmin,l,c,r);
      }

  for (int s=0; s<ns; s++) {
    string st=format("%s%03d.png",argv[2],s+rmin);
    imp(normaliza(sai[s]),st);
  }
}
```


circle1.jpg, sai1-ed.png, sai1-hb056.png, sai1-hb057.png, sai1-hb058.png, sai1-hb059.png, sai1-hb060.png, sai1-b2.png

```
>kcek houcirb sai1-ed.png sai1-hb 56 60
>houcirb2 sai1-ed.png sai1-b2.png 56 60
minimo=-124.000000 s=2 raio=58 l=64 c=78
```

(PSI3472-2019 Aula 4 Exercício 1) Execute houcirb.cpp para calcular transformada de Hough para círculos nas imagens doisptos.png, tresptos.png, circulo.png e sai1-ed.png. Em cada caso, escolha adequadamente rmin e rmax para observar os centros dos círculos.

(PSI3472-2019 Aula 4 Exercício 2) Modifique houcirb.cpp para detectar um único círculo nas imagens tresptos.png, circulo.png e sai1-ed.png. Pinte o círculo detectado em vermelho, como na imagem sai1-b2.png.


tresptos.ppm, circulo.ppm, sai1-ed.ppm

Como traçar círculos usando função do OpenCV

```
void circle(Mat& img, Point center, int radius, const Scalar& color, int thickness=1, int li-
neType=8, int shift=0)
Ex: Mat_<COR> a; ...; circle(a,Point(co,lo),r,Scalar(0,0,255));
```

Como traçar círculos usando função do Cekeikon

```
void circulo(Mat_<COR>& a, int l, int c, int r, COR cor=COR(0,0,0), int t=1);
Ex: circulo(a,lo,co,r,COR(0,0,255));
```

Exemplo de uso de função converte

```
Mat_<COR> c(100,100,COR(0,0,255)); Mat_<FLT> f; converte(c,f);
```

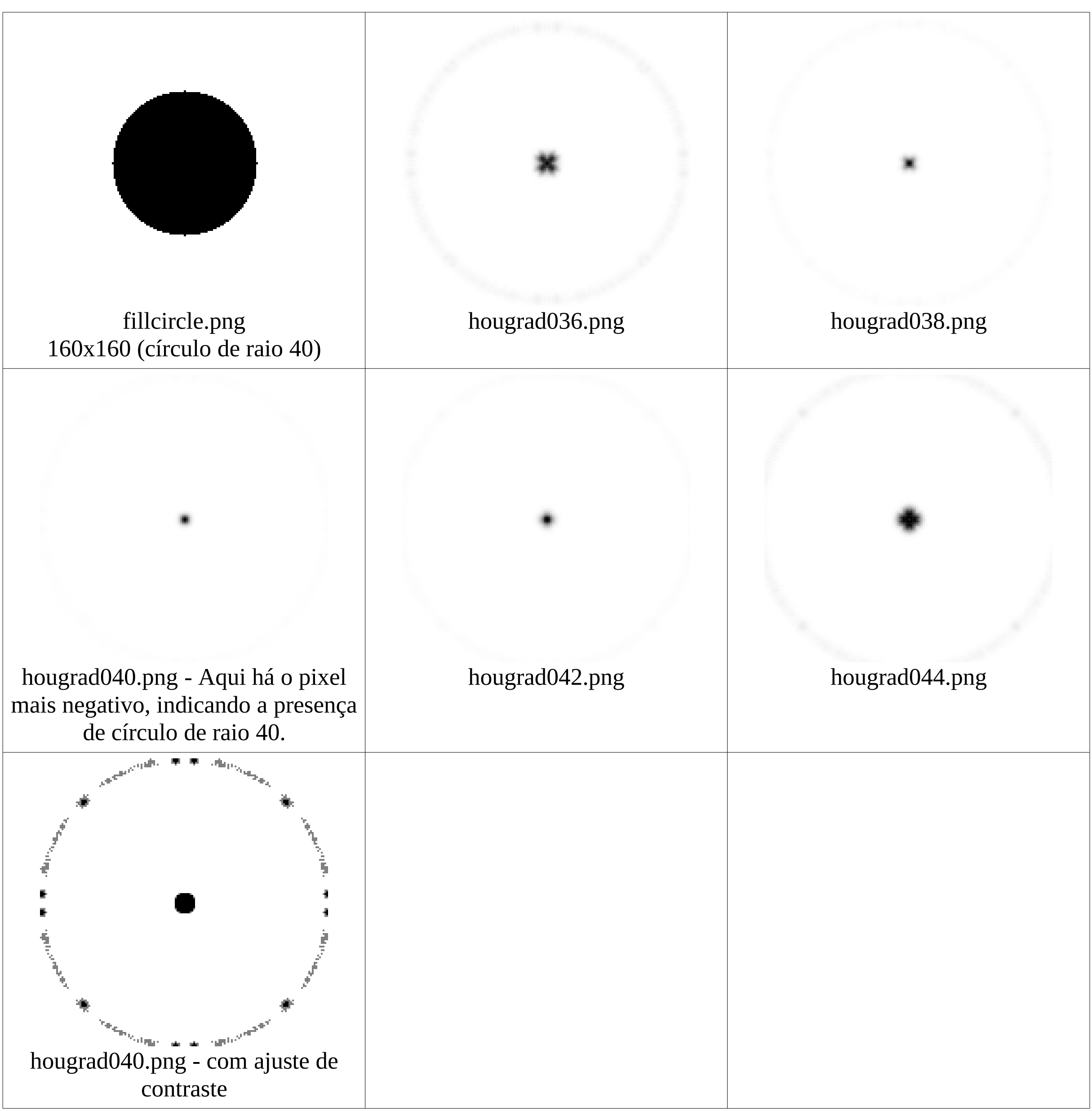
RODANDO

Usando HoughCircles do OpenCV2:

```
// houghcircle.cpp - pos2013
#include <cekeikon.h>
using namespace Morphology;

int main(int argc, char** argv)
{ if (argc!=3) erro("HoughCircle ent.pgm sai");
  Mat_<GRY> ent; le(ent,argv[1]);
  GaussianBlur( ent, ent, Size(9, 9), 2, 2 );

  Mat_<GRY> b;
  Canny(ent,b,200,100);
  imp(-b,string(argv[2])+"-ed.png");

  vector<Vec3f> circles;
  HoughCircles(ent,circles,CV_HOUGH_GRADIENT,2,ent.rows/8,200,100,40,80);
  Mat_<COR> a; converte(ent,a);
  for (unsigned i = 0; i<circles.size(); i++) {
    Point center(round(circles[i][0]), round(circles[i][1]));
    int radius = round(circles[i][2]);
    circle( a, center, 3, Scalar(0,255,0), -1, 8, 0 );
    circle( a, center, radius, Scalar(0,0,255), 3, 8, 0 );
  }
  imp(a,string(argv[2])+"-ci.png");
}
```

Nota: A função OpenCV que detecta círculos pela transformada de Hough também não funciona direito. Aparentemente, detecta primeiro as arestas da imagem antes de detectar círculos. A implementação (tanto na versão 2 como na versão 3) visa eficiência de espaço (pouco uso de memória) e não acuracidade. O manual diz: "For sake of efficiency, OpenCV implements a detection method slightly trickier than the standard Hough Transform: The Hough gradient method, which is made up of two main stages. The first stage involves edge detection and finding the possible circle centers and the second stage finds the best radius for each candidate center. For more details, please check the book Learning OpenCV or your favorite Computer Vision bibliography".

Transformada de Hough para detectar círculos usando módulo e direção de gradiente.



fillcircle.png
160x160 (círculo de raio 40)

hougrad036.png

hougrad038.png

hougrad040.png - Aqui há o pixel mais negativo, indicando a presença de círculo de raio 40.

hougrad042.png

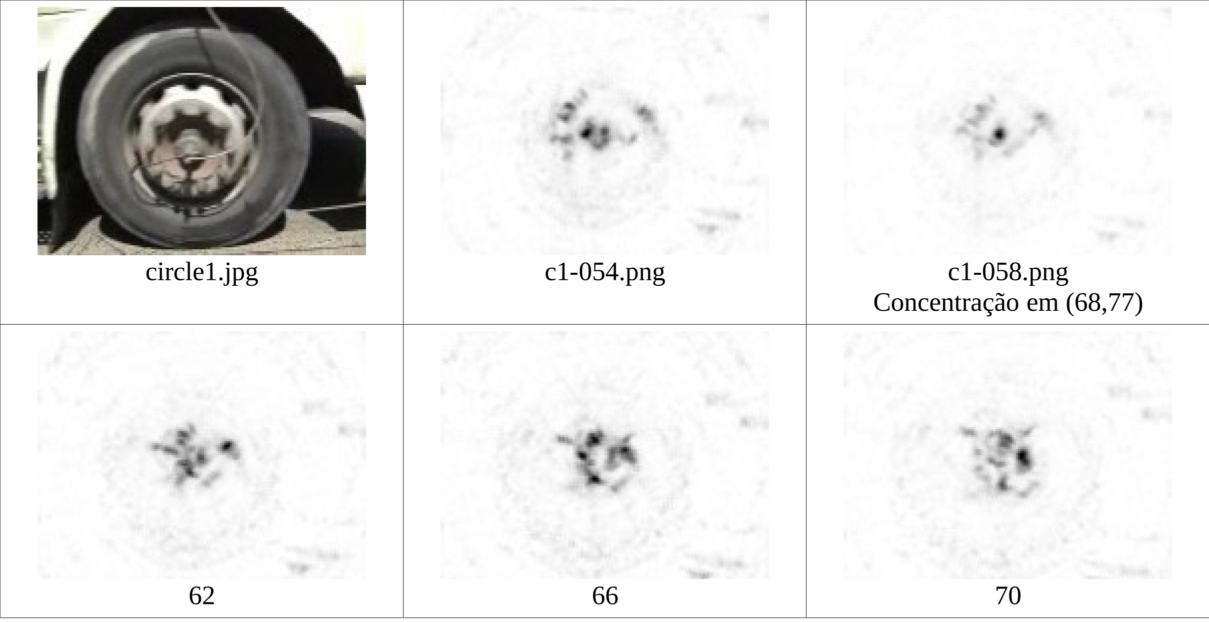
hougrad044.png

hougrad040.png - com ajuste de contraste

Rodei:

```
>fillcircle
>houcirgr fillcircle.png houcirgr 30 50
```

Transformada de Hough para detectar círculos usando módulo e direção de gradiente.



circle1.jpg	c1-054.png	c1-058.png Concentração em (68,77)
62	66	70

Detecção de círculos em imagens coloridas
Devolve os circulos devolve num vetor, junto com o valor acumulado.
Devolve circulos com acumulador<limiar ou max_circulos se houver muitos circulos

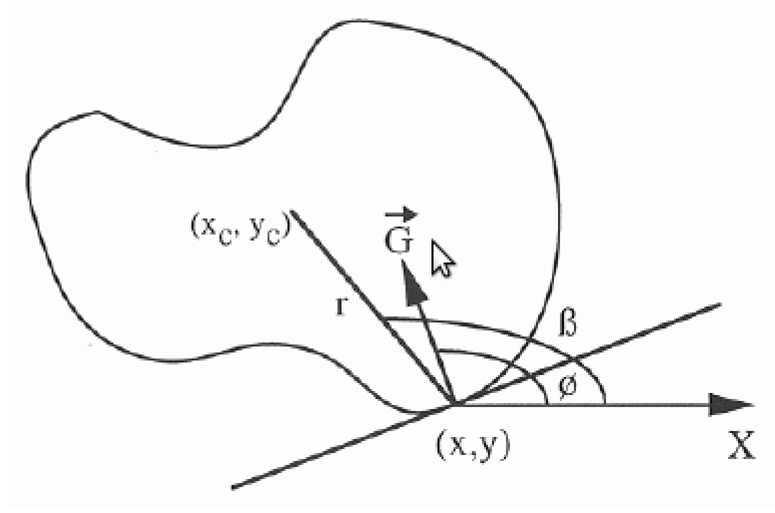
```
//houcirc.cpp - 2019
//Detecta circulos e os devolve num vetor, junto com valor acumulado.
//Devolve circulos com acumulador<limiar ou max_circulos se houver muitos circulos
#include <cekeikon.h>

void subtrai(Mat_<FLT>& b, int l, int c, int borraoespacial, float magn) {
  for (int l2=l-borraoespacial; l2<=l+borraoespacial; l2++)
    for (int c2=c-borraoespacial; c2<=c+borraoespacial; c2++)
      if (0<=l2 && l2<b.rows && 0<=c2 && c2<b.cols)
        b(l2,c2) -= magn;
}

void houghCir(Mat_<COR> _ent,
              vector<Vec3i>& circulos, // x, y, raio
              vector<float>& acumulado,
              float limiar, int max_circulos, // Devolve circulos com acumulador<limiar ou
                                     // max_circulos se houver muitos circulos.
              double min_dist,
              int rmin,
              int rmax,
              int borraoespacial=1,
              double GaussDev=2.0,
              double pesormax=1.0,
              double pesormin=1.0) {
  Mat_<COR> ent=_ent.clone();
  int nl=ent.rows;
  int nc=ent.cols;
  int ns=rmax-rmin+1;

  vector< Mat_<FLT> > acc(ns);
  for (int i=0; i<ns; i++) {
    acc[i].create(nl,nc);
    acc[i].setTo(0.0); //comecou com zero
  }

  GaussianBlur(ent,ent,Size(0,0),GaussDev,GaussDev);

  Mat_<CPX> grad=gradienteScharr(ent,true); // y para cima
  Mat_<FLT> magn=modulo(grad);
  Mat_<CPX> vers=versor(grad);

  for (int l=0; l<ent.rows; l++) {
    for (int c=0; c<ent.cols; c++)
      if (magn(l,c)>0.001) {
        // Traca reta em acc, a partir do centro (l,c).
        for (unsigned s=0; s<acc.size(); s++) {
          int r=s+rmin;
          int l2=round(l-imag(vers(l,c))*r);
          int c2=round(c+real(vers(l,c))*r);
          subtrai(acc[s],l2,c2,borraoespacial,magn(l,c));
          l2=round(l+imag(vers(l,c))*r);
          c2=round(c-real(vers(l,c))*r);
          subtrai(acc[s],l2,c2,borraoespacial,magn(l,c));
        }
      }
  }

  // Para cada pixel, acha a escala (raio) que tem o menor valor no acc.
  // Preenche minimo e escala
  Mat_<FLT> menoracc(ent.size());
  Mat_<int> menorescala(ent.size());
  for (unsigned i=0; i<ent.total(); i++) {
    float minval=0; int minind=0;
    for (int s=0; s<ns; s++)
      if (acc[s](i)<minval) { minval=acc[s](i); minind=s; }
    menoracc(i)=minval;
    menorescala(i)=minind;
  }

  //Pode dar mais peso para raios menores ou maiores
  if (pesormax!=1.0 || pesormin!=1.0)
    for (unsigned i=0; i<ent.total(); i++) {
      double peso = pesormin + ( (pesormax-pesormin) * (ns-1-menorescala(i)) / (ns-1) );
      // ns-1             pesormax
      // menorescala      peso
      // 0                pesormin
      //
```

```
      // ns-1               pesormax-pesormin
      // ns-1-menorescala  peso-pesormin
      //
      // (peso-pesormin)*(ns-1) = (pesormax-pesormin)*(ns-1-menorescala)
      //
      // peso = (pesormax-pesormin)*(ns-1-menorescala)/(ns-1) + pesormin
      menoracc(i)*=peso;
    }

  // Normaliza menoracc para 0..1
  Mat_<FLT> menoraccnormal=normaliza(menoracc);

  // Chama vector<Point> kmin(Mat_<FLT> a, int k, double d=0.0) para achar os k minimos.
  // Os pontos de a com valor 1 nao sao levados em conta.
  vector<Point> centros=kmin(menoraccnormal,max_circulos,min_dist);

  for (int i=0; i<centros.size(); i++) {
    Point c=centros[i];
    float a=menoracc(c);
    if (menoracc(c)<limiar) {
      int r=menorescala(c)+rmin;
      Vec3f v(c.x,c.y,r); circulos.push_back(v);
      acumulado.push_back(a);
    }
  }
}

int main(int argc, char** argv) {
  const char* keys =
    " curto| longo    |default| explicacao\n"
    " -l   | -limiar  | -15   | Devolve circulos com acc<limiar\n"
    " -c   | -maxcirc | 20    | Numero maximo de circulos\n"
    " -d   | -mindist | 20    | Distancia minima entre dois circulos\n"
    " -min | -rmin    | 20    | Raio minimo do circulo\n"
    " -max | -rmax    | 80    | Raio maximo do circulo\n"
    " -b   | -borrao  |  1    | Borrao espacial no espaco de Hough\n"
    " -g   | -gaussd  |  2    | Desvio padrao do filtro gaussiano\n"
    " -pmin| -pmin    |  1    | Peso do circulo de raio minimo\n"
    " -pmax| -pmax    |  1    | Peso do circulo de raio maximo\n"
    " -nl  | -nlinhas |  0    | Redimensiona para nl. 0=nao redimensiona\n"
    " -nc  | -ncolunas|  0    | Redimensiona para nc. 0=nao redimensiona\n";

  if (argc<3) {
    printf("HouCirC ent.ppm sai.ppm [opcoes]\n");
    printf("%s",keys);
    erro("Erro: Numero de argumentos invalido");
  }

  ArgComando cmd(argc,argv);
  string nomeent=cmd.getCommand(0);
  if (nomeent=="") erro("Erro: Nao especificou imagem de entrada");
  string nomesai=cmd.getCommand(1);
  if (nomesai=="") erro("Erro: Nao especificou imagem de saida");

  float limiar=cmd.getFloat("-l","-limiar",-15);
  int max_circulos=cmd.getInt("-c","-maxcirc",20);
  double min_dist=cmd.getDouble("-d","-mindist",20);
  int rmin=cmd.getInt("-min","-rmin",20);
  int rmax=cmd.getInt("-max","-rmax",80);
  int borrao=cmd.getInt("-b","-borrao",1);
  double gaussd=cmd.getDouble("-g","-gaussd",2);
  double pesormin=cmd.getDouble("-pmin","-pmin",1);
  double pesormax=cmd.getDouble("-pmax","-pmax",1);
  int nl=cmd.getInt("-nl","-nlinhas",0);
  int nc=cmd.getInt("-nc","-ncolunas",0);
  cmd.leuTodos();

  Mat_<COR> ent; le(ent,nomeent);
  if (nl>0 || nc>0)
    resize(ent,ent,Size(nc,nl),INTER_AREA);

  vector<Vec3i> circulos; // x, y, raio
  vector<float> acumulado;
  houghCir(ent, circulos, acumulado,
           limiar, max_circulos, min_dist, rmin, rmax,
           borrao, gaussd, pesormin, pesormax);

  for (int i=0; i<circulos.size(); i++) {
    Point center(circulos[i][0], circulos[i][1]);
    int radius = circulos[i][2];
    // draw the circle center
    circle( ent, center, 3, Scalar(0,255,0), -1, 8, 0 );
    // draw the circle outline
    circle( ent, center, radius, Scalar(0,0,255), 3, 8, 0 );
```

```
  }
  imp(ent,nomesai);
}
```

(PSI3472-2019 Aula 4 Exercícios 3 e 4) Resolva EP1 desta disciplina, usando a função houghCir acima (ou HoughCircles do OpenCV, se preferir).

Transformada de Hough generalizada



H

$\phi_1 = 0$	$(r,\beta)_{1_1}$	$(r,\beta)_{1_2}$	$\cdots$	$(r,\beta)_{1_{n_1}}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
ϕ_j	$(r,\beta)_{j_1}$	$(r,\beta)_{j_2}$	$\cdots$	$(r,\beta)_{j_{n_1}}$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\phi_k = \pi$	$(r,\beta)_{k_1}$	$(r,\beta)_{k_2}$	$\cdots$	$(r,\beta)_{k_{n_1}}$

Pré-processamento:
a) Dada uma forma *F* que se deseja encontrar, defina arbitrariamente um ponto central (x_c, y_c), mais ou menos no centro da forma.
b) Crie uma vetor de listas *H* indexada por ϕ de 0 até 180° (evidentemente, discretizada em algum passo).
c) Para cada ponto da borda (x, y), calcule o gradiente $\vec{G}$, o ângulo do gradiente ϕ, a distância r entre os pontos (x, y) e (x_c, y_c) e o ângulo β do segmento de reta entre os pontos (x, y) e (x_c, y_c). Depois, acrescente o par (r, β) no vetor *H*, na lista de indexada por ϕ.

Cálculo da transformada de Hough generalizada:
Dada uma imagem *I*, calcule o gradiente em todos os pixels.
Para todos os pixels (x, y) da imagem com magnitude de gradiente "suficientemente grande":
    Calcule o ângulo do gradiente ϕ;
    Vá na tabela *H*, na lista indexada por ϕ;
    Para todos os pares (r, β) daquela lista:
        Usando ϕ, r e β, calcule o centro da forma (x_c, y_c) indicada pelo pixel (x, y);
        Subtraia um (ou some um) no espaço de Hough, no ponto (x_c, y_c);
Os locais da imagem *I* onde aparecem a forma *F* deve ter picos negativos no espaço de Hough.

É possível modificar este algoritmo para obter invariância à rotação e/ou escala.

O site abaixo traz uma implementação de transformada de Hough generalizado, para detectar a chave em qualquer rotação (só detecta numa única escala).

http://www.itriacasa.it/generalized-hough-transform/

No meu computador, deixei esse programa em: ~/algpi/hough/generalizado

Rode:
ght Img_01.png
ght Img_02.png
ght Img_03.png

OpenCV3 parece que consegue usar gradiente na transformada de Hough
OpenCV3 possui classe GeneralizedHough

Referências:

[Gonzalez2002] R. C. Gonzalez, R. E. Woods, “Digital Image Processing, Second Edition,” Prentice-Hall, 2002.

[Ballard1981] D. H. Ballard, “Generalizing the Hough Transform to Detect Arbitrary Shapes,” *Pattern Recognition*, vol. 13, no. 2, pp. 111-122, 1981.

[hmc] http://fourier.eng.hmc.edu/e161/lectures/hough/node6.html

Wikipedia

Para descobrir se dois casamentos de SIFT (ou SURF) pertencem a um mesmo objeto.

Cada ponto-chave K do SIFT consiste de $[K_x, K_y, K_r, K_s]$, respectivamente coordenadas x e y, ângulo de rotação r e escala s (além dos 128 descritores).

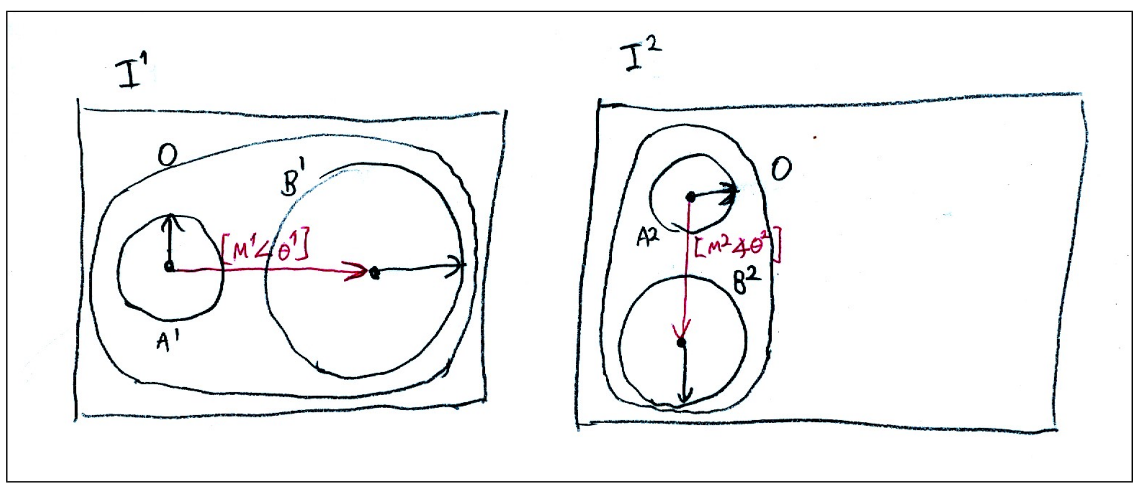


Figura 1

Vamos supor que um objeto O possui dois pontos-chaves A e B (figura 1). Objeto O aparece nas imagens I^1 e I^2 em diferentes posições, ângulos e escalas. Os pontos-chaves A e B tornam-se:

$A^1 = [A_x^1, A_y^1, A_r^1, A_s^1]$ e $B^1 = [B_x^1, B_y^1, B_r^1, B_s^1]$ na figura I^1 e

$A^2 = [A_x^2, A_y^2, A_r^2, A_s^2]$ e $B^2 = [B_x^2, B_y^2, B_r^2, B_s^2]$ na figura I^2.

Neste caso, devem valer as seguintes igualdades:

1) As razões entre as escalas dos pontos-chaves A e B nas duas imagens devem ser os mesmos:

$$\frac{A_s^2}{A_s^1} = \frac{B_s^2}{B_s^1} = s.$$

2) As diferenças dos ângulos de rotação dos pontos-chaves A e B nas duas imagens devem ser iguais:

$$A_r^2 \ominus A_r^1 = B_r^2 \ominus B_r^1 = r.$$

onde $\ominus$ é subtração de ângulos (o resultado deve estar entre 0 e 2π).

Além disso, vamos denotar a diferença vetorial entre os centros dos dois pontos-chaves na forma polar como:

$$\begin{cases} [M^1 \measuredangle \theta^1] = [B_x^1, B_y^1] - [A_x^1, A_y^1] \\ [M^2 \measuredangle \theta^2] = [B_x^2, B_y^2] - [A_x^2, A_y^2] \end{cases}$$

onde M é a magnitude e θ é o ângulo do vetor-diferença. Então, valem as seguintes igualdades:

3) $M^2/M^1 = s$, onde s é a razão de escalas da equação 1.

4) $\theta^2 \ominus \theta^1 = r$, onde r é a diferença de ângulos da equação 2.

Utilizando as quatro igualdades acima, é possível agrupar casamentos em grupos. Então, os casamentos falsos que não pertencem a nenhum grupo podem ser descartados.

Hough para reconhecer objetos depois de SIFT

vetorial ou matricial.

Calculando Hough generalizado após SIFT para localizar objetos:

```
>sift Find_obj olho.tga lennag.tga olho.ppm
```

olho.tga

lennag.tga

olho.ppm