[PSI3471 aula 5 - Início]

Casamento de modelo (template matching)

[Nota para mim: Preciso reorganizar esta apostila. Trocar exemplo 1D por template maior. Mostrar que quanto mais parecido a instância de A com Q, maior a correlação em P.]
[Nota para mim: A partir do OpenCV 4.4, é possível passar máscara para matchTemplate.]

As imagens usadas nesta aula podem ser baixadas em Linux com os comandos:

```
$ wget -U 'Firefox/50.0' http://www.lps.usp.br/hae/apostila/tmatch.zip
$ unzip tmatch.zip
```

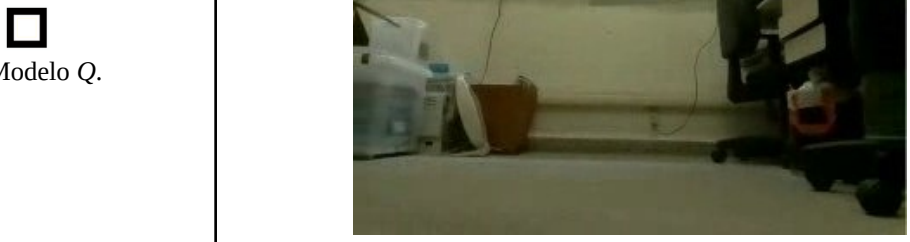
Além disso, os programas desta aula usam as funções de procimagem.h:

```
//procimagem.h 2024
#include <opencv2/opencv.hpp>
#include <iostream>
#include <float.h>
using namespace std;
using namespace cv;

void erro(string s1="") {
  cerr << s1 << endl;
  exit(1);
}

Mat_<float> filtro2d(Mat_<float> ent, Mat_<float> ker, int borderType=BORDER_DEFAULT)
{ Mat_<float> sai;
  filter2D(ent,sai,-1,ker,Point(-1,-1),0.0,borderType);
  return sai;
}

Mat_<Vec3f> filtro2d(Mat_<Vec3f> ent, Mat_<float> ker, int borderType=BORDER_DEFAULT)
{ Mat_<Vec3f> sai;
  filter2D(ent,sai,-1,ker,Point(-1,-1),0.0,borderType);
  return sai;
}

Mat_<float> dcReject(Mat_<float> a) { // Elimina nivel DC (subtrai media)
  Mat_<float> b=a-mean(a)[0];
  return b;
}

Mat_<float> dcReject(Mat_<float> a, float dontcare) {
  // Elimina nivel DC (subtrai media) com dontcare
  Mat_<uchar> naodontcare = (a!=dontcare);
  Mat_<uchar> simdontcare = (a==dontcare);
  Scalar media=mean(a,naodontcare);
  Mat_<float> b=a.clone();
  subtract(b,media[0],b,naodontcare);
  subtract(b,dontcare,b,simdontcare);
  return b;
}

Mat_<float> somaAbsDois(Mat_<float> a) { // Faz somatoria absoluta da imagem dar dois
  double soma = sum(abs(a))[0];
  Mat_<float> b = a / (soma/2.0);
  return b;
}

Mat_<float> matchTemplateSame(Mat_<float> a, Mat_<float> q, int method,
  float backg=0.0) {
  Mat_<float> p{ a.size(), backg };
  Rect rect{ (q.cols-1)/2, (q.rows-1)/2, a.cols-q.cols+1, a.rows-q.rows+1};
  Mat_<float> roi{ p, rect };
  matchTemplate(a, q, roi, method);
  return p;
}
```

Programa P1: Arquivo procimagem.h desta aula.

1. Introdução

Casamento de modelo (ou casamento de máscara, “template matching” em inglês) é uma técnica usada para achar as instâncias de um modelo (ou máscara ou *template image*) *Q* dentro de uma imagem *A* a ser analisada (ou imagem onde fazer a pesquisa ou *search image*). A figura 1 ilustra uma aplicação desta técnica.



(a) Modelo *Q*.

(b) Imagem analisada *A* com modelo *Q* localizado.

Figura 1: Exemplo de casamento de modelo.

Casamento de modelo tradicionalmente utiliza filtro linear (ou convolução). Casamento de modelo usando filtro linear possui várias propriedades interessantes:

1) É possível calcular rapidamente o casamento usando FFT.
2) É possível obter invariância a brilho.
3) É possível obter invariância a brilho e contraste modificando (um pouco) o filtro linear.
4) É possível especificar no modelo pixels cujos valores não têm importância (“don’t care”).

2. Caso binário e unidimensional:

Para entender o funcionamento de casamento de modelo, vamos tentar resolver o seguinte problema. São dados vetores binários *a* e *q*. Gostaríamos de detectar as ocorrências de *q* dentro de *a*, usando filtro linear. Essas ocorrências estão marcadas em amarelo:

```
a = [ 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0]
q = [ 0, 1, 1]
```

Se simplesmente aplicarmos filtro linear em *a* usando *q* como peso, não dá certo:[1]

```
p = filtro2d(a,q,BORDER_REPLICATE)
p = [ 1, 1, 0, 1, 1, 1, 2, 2, 1, 1, 2, 1, 0]
```

Filtro linear até detectou as duas ocorrências de 011 (marcadas em verde) gerando valores altos nessas posições. Mas há um falso negativo (marcado em vermelho). Note que calcular “média aritmética ponderada” do filtro linear é o mesmo que calcular “produto escalar” entre os vetores. O produto escalar entre *q*=[011] e os vetores [011] e [111] dão o mesmo resultado 2.

[1]Estou chamando a função *filtro2d* com “BORDER_REPLICATE” para que o filtro considere que os pixels fora do domínio do vetor são iguais aos pixels mais próximos dentro do domínio.

Como podemos detectar corretamente as ocorrências de *q* em *a*? A solução é subtrair a média de *q* de cada um dos elementos de *q*. A média de *q* é (0+1+1)/3 = 0.67. Subtraindo 0.67 de cada elemento de *q* obtemos *q2*:

```
q2=dcReject(q)
q2=[-0.67, 0.33, 0.33]
```

Esta operação chama-se *correção de média*. Coloquei a função *dcReject* dentro do arquivo *procimagem.h* que efetua a correção de média. Agora, vamos filtrar o vetor *a* usando o peso *q2*:

```
p2=filtro2d(a,q2,BORDER_REPLICATE)
p2=[0.3, 0.3, -0.7, 0.3, 0.3, -0.3, 0.7, 0.0, -0.3, -0.3, 0.7, -0.3, -0.7]
```

Desta vez o filtro detectou corretamente as duas ocorrências de 011 dentro do vetor *a*, gerando os maiores valores apenas nessas posições.

A operação que obtivemos é invariante a brilho, isto é, se somarmos uma constante no vetor *a*, continuamos obtendo a mesma saída. No exemplo abaixo, somamos 10 ao vetor *a*, mas obtemos o mesmo resultado.

```
a3=a+10
a3=[ 10, 11, 10, 10, 11, 10, 11, 11, 11, 10, 11, 11, 10]
p3=filtro2d(a3,q2,BORDER_REPLICATE)
p3= [0.3, 0.3, -0.7, 0.3, 0.3, -0.3, 0.7, 0.0, -0.3, -0.3, 0.7, -0.3, -0.7]
```

O único defeito deste método é que os casamentos perfeitos estão dando correlação 0.7. Gostaríamos que a saída estivesse dentro de um intervalo conhecido, por exemplo, [-1, +1]. A função *somaAbsDois* faz a soma absoluta dos elementos do modelo dar dois. Usando esta função, se a imagem de entrada vai de 0 a 1, a saída do filtro linear estará limitada a [-1, +1]. E se a imagem de entrada vai de 0 a 255, a saída estará limitada a [-255, +255]:

```
q4=somaAbsDois(dcReject(q))
q4= [-1, 0.5, 0.5]
p4=filtro2d(a,q4,BORDER_REPLICATE)
p4= [0.5, 0.5, -1, 0.5, 0.5, -0.5, 1, 0, -0.5, -0.5, 1, -0.5, -1]

a = [  0,   1,  0,   0,   1,    0, 1, 1,    1,    0, 1,    1,  0]  0
```

Detectamos as duas ocorrências de 011 no vetor *a*, com saídas exatamente iguais a um. As saídas -1 identificam locais onde ocorre a instância negativa, isto é, 100.

Em *procimagem.h*:

A função dcReject(*q*) subtrai a média de *q* de todos os pixels de *q* (faz correção de média).

A função somaAbsDois(*b*) divide todos os pixels de *b* pela somatória dos valores absolutos de *b* dividido por 2.

O programa abaixo implementa os exemplos 1D que escrevemos acima.

```
//match1d.cpp 2024
#include "procimagem.h"

int main()
{ Mat_<float> a = ( Mat_<float>(1,13) <<
                    0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0); cout << a << endl;
  Mat_<float> q = ( Mat_<float>(1,3) << 0, 1, 1 ); cout << q << endl;
  Mat_<float> p=filtro2d(a,q,BORDER_REPLICATE); cout << p << endl;

  Mat_<float> q2=dcReject(q); cout << q2 << endl;
  Mat_<float> p2=filtro2d(a,q2,BORDER_REPLICATE); cout << p2 << endl;

  Mat_<float> a3=a+10; cout << a3 << endl;
  Mat_<float> p3=filtro2d(a3,q2,BORDER_REPLICATE); cout << p3 << endl;

  Mat_<float> q4=somaAbsDois(dcReject(q)); cout << q4 << endl;
  Mat_<float> p4=filtro2d(a,q4,BORDER_REPLICATE); cout << p4 << endl;
}
```

Programa 1: match1d.cpp.

A saída obtida é:

```
a = [0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0]
q = [0, 1, 1]
p = [1, 1, 0, 1, 1, 1, 2, 2, 1, 1, 2, 1, 0]

q2 = [-0.667, 0.333, 0.333]
p2 = [0.333, 0.333, -0.667, 0.333, 0.333, -0.333, 0.666, 0, -0.333, -0.333, 0.667, -0.333, -0.667]

a3 = [10, 11, 10, 10, 11, 10, 11, 11, 11, 10, 11, 11, 10]
p3 = [0.333, 0.333, -0.667, 0.333, 0.333, -0.333, 0.666, 0, -0.333, -0.333, 0.667, -0.333, -0.667]

q4 = [-1, 0.5, 0.5]
p4 = [0.5, 0.5, -1, 0.5, 0.5, -0.5, 1, 0, -0.5, -0.5, 1, -0.5, -1]
```

Exercício: Modifique o programa 1 para que procure o modelo

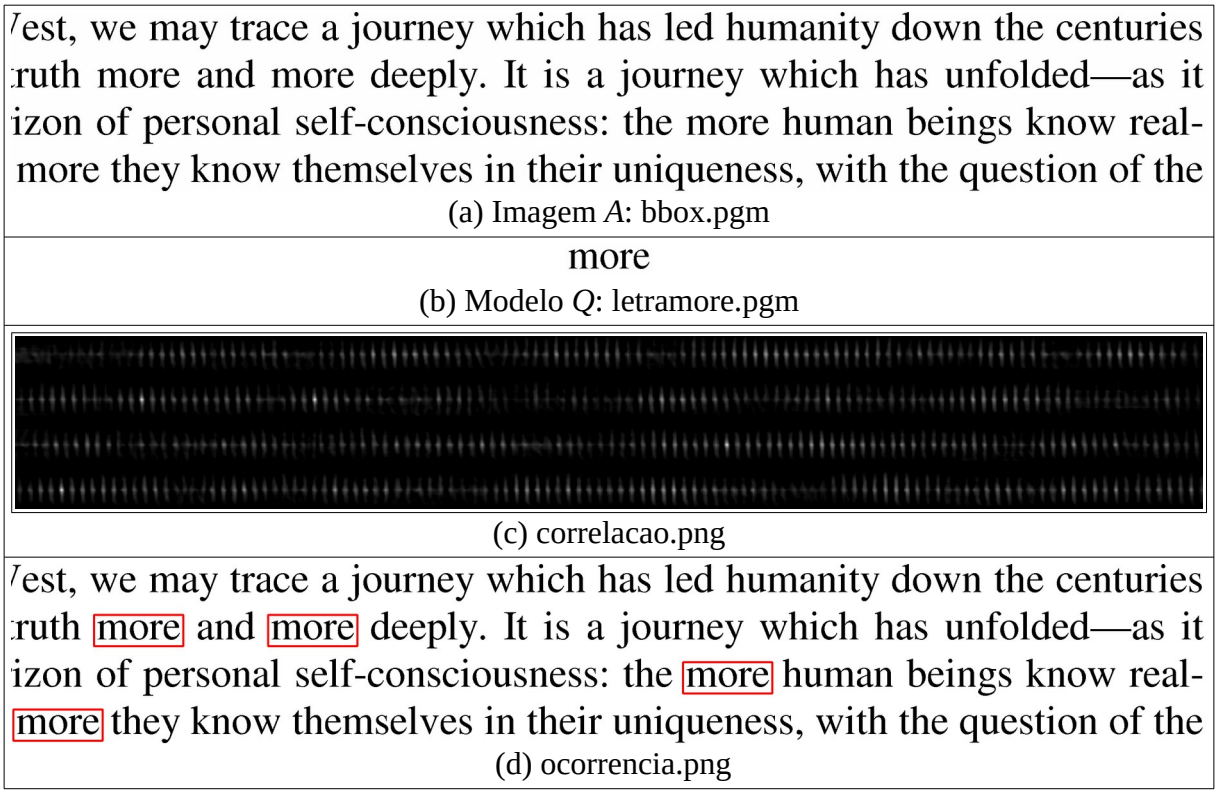
```
q = [0, 1, 0, 1, 0]
```

no vetor

```
a = [0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0]
```

3. Caso binário bidimensional:

Podemos aplicar as ideias vistas na seção anterior para detectar um modelo binário Q (letramore.pgm da figura 2) numa imagem binária A (bbox.pgm da figura 2). O programa 2 (match2d.cpp) faz isso e a sua saída está nas figuras 2c e 2d.



(a) Imagem A: bbox.pgm

(b) Modelo Q: letramore.pgm

(c) correlacao.png

(d) ocorrencia.png

Figura 2: Detecção da palavra “more” no documento usando filtro linear.

```
 1 //match2d.cpp 2024
 2 #include "procimagem.h"
 3 int main() {
 4   Mat_<float> a=imread("bbox.pgm",0); if (a.total()==0) erro("Erro"); a=a/255.0;
 5   Mat_<float> q=imread("letramore.pgm",0); if (q.total()==0) erro("Erro"); q=q/255.0;
 6   q=somaAbsDois(dcReject(q));
 7   Mat_<float> p=filtro2d(a,q);
 8   imwrite("correlacao.png",255*p);
 9
10   Mat_<Vec3f> d; cvtColor(a,d,COLOR_GRAY2BGR);
11   for (int l=0; l<a.rows; l++)
12     for (int c=0; c<a.cols; c++)
13       if (p(l,c)>=0.999)
14         rectangle(d,Point(c-109,l-38),Point(c+109,l+38),Scalar(0,0,1),3);
15   imwrite("ocorrencia.png",255*d);
16 }
```

Programa 2: match2d.cpp.

Para facilitar, vamos convencionar que sempre trabalharemos com imagens em ponto flutuante com valores entre 0 (preto) e 1 (branco). Vamos dividir imagens por 255 após a leitura e multiplicar as imagens por 255 antes da impressão.

Nas linhas 4 e 5, o programa 2 lê as imagens a e q como matrizes ponto flutuante e divide por 255 para que os pixels estejam entre 0 e 1. A linha 6 executa “q=somaAbsDois(dcReject(q))”

que vimos ser necessário para fazer casamento de modelo usando filtro linear. A linha 7 aplica o filtro e a linha 8 imprime a matriz resultante (com valores entre 0 e 255) como imagem em níveis de cinza.

As linhas 10-15 pintam de retângulo vermelho a região em torno do casamento (correlação 1.0). Na linha 13, são calculados os picos correlação acima de 0,999.

Nota: O program filter2D fica consideravelmente mais rápido na OpenCV3 do que em OpenCV2.

```
$compila match2d -ocv (linka com OpenCV2 – demora 0.15s)
$compila match2d -ocv -v3 (linka com OpenCV3 – demora 0.08s).
$compila.sh match2d (linka com OpenCV4 – demora 0.22s).
```

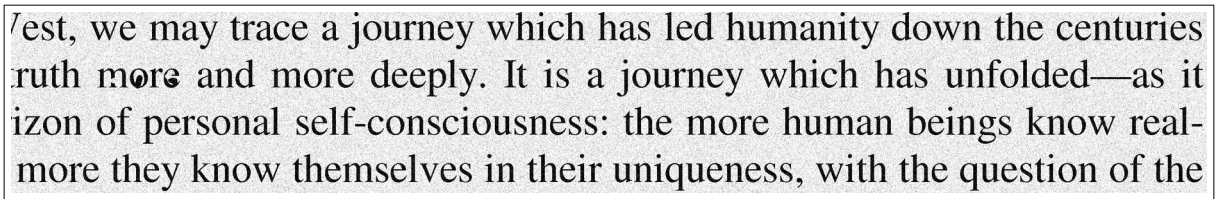
Obs: Alguém poderia perguntar: Por que usar filtro linear, se poderia escrever uma função simples que conta, para cada posição da janela móvel, o número de pixels iguais (ou diferentes) entre o modelo *Q* e os pixels de *A* dentro da janela móvel?

- Resposta 1: Porque filtro linear pode ser calculado rapidamente usando FFT, principalmente para modelos grandes.
- Resposta 2: Porque se imagem *A* for níveis de cinza, a matriz obtida *P* será independente do brilho e proporcional ao contraste.
- Resposta 3: Porque se imagem *A* for níveis de cinza, a matriz obtida *P* irá indicar também as instâncias “parecidas” com o modelo *Q*.

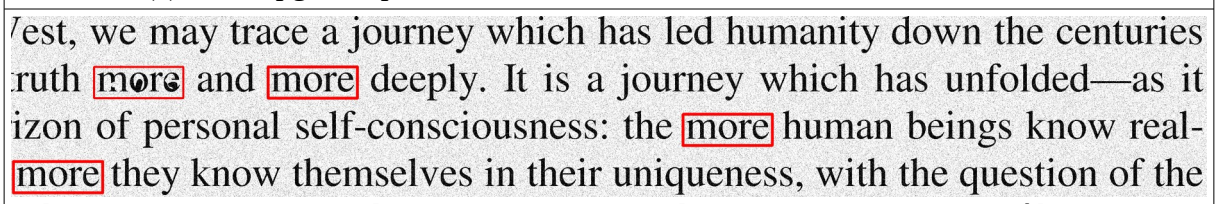
Vamos aguardar a aula avançar um pouco mais para compreender estas respostas.

Exercício: Escreva manualmente (isto é, sem usar funções prontas das bibliotecas) um filtro que calcula, para cada posição da janela móvel, média da diferença absoluta entre o conteúdo da janela e o modelo (vamos denominar o programa de *maematch.cpp*). Use esse programa para localizar “letramore.bmp” dentro de “bbox.bmp”. Compare o tempo de processamento do programa 2 e com o do seu programa.
Resposta: O programa escrito manualmente demora ≈1,5 minutos, enquanto que o programa escrito usando função pronta do OpenCV demora ≈0,15 segundos.

Usando filtro linear, casamento de modelos também consegue detectar instâncias ruidosas. Veja a bbox2.pgm (figura 4a) onde foram inseridas ruído e sujeiras na imagem bbox.pgm e uma palavra “more” foi corrompida. Usando limiar 0,999, não é mais possível detectar as palavras “more”. Baixando o limiar para 0,7, o programa consegue detectar as instâncias mesmo com sujeira.



(a) bbox2.pgm: O primeiro “more” foi alterado e ruídos foram inseridos.



(b) ocorrencia2.png: Com limiar 0,7, conseguiu achar “more” mesmo com ruído e sujeira.

Figura 4: Baixando limiar, foi possível detectar instâncias ruidosas de “more”.

```
1  //match2d2.cpp 2024
2  #include "procimagem.h"
3  int main() {
4    Mat_<float> a=imread("bbox2.pgm",0); if (a.total()==0) erro("Erro"); a=a/255.0;
5    Mat_<float> q=imread("letramore.pgm",0); if (q.total()==0) erro("Erro"); q=q/255.0;
6    q=somaAbsDois(dcReject(q));
7    Mat_<float> p=filtro2d(a,q);
8    imwrite("correlacao2.png",255*p);
9
10   Mat_<Vec3f> d;
11   cvtColor(a,d,COLOR_GRAY2BGR); // OpenCV4
12   for (int l=0; l<a.rows; l++)
13     for (int c=0; c<a.cols; c++)
14       if (p(l,c)>0.7)
15         rectangle(d,Point(c-109,l-38),Point(c+109,l+38),Scalar(0,0,1),3);
16   imwrite("ocorrencia2.png",255*d);
17 }
```

Programa P2: Baixando o limiar para 0,7, é possível detectar palavra “more” mesmo na imagem ruidosa *bbox2.pgm*.

4. Pixels indiferentes ("don't care"):

Veja a imagem bbox3.pgm (figura 4a), onde 3 das 4 palavras "more" foram tão danificadas que é impossível encontrá-las simplesmente baixando o limiar. Baixando limiar para 0,64 (programa P3) ainda não foi possível encontrar todas as palavras "more" danificadas mas já apareceu um falso positivo: encontrou erroneamente "more" na palavra "uniqueness" (figura 4d).

```
//match2d3.cpp 2024
#include "procimagem.h"
int main() {
  Mat_<float> a=imread("bbox3.pgm",0); if (a.total()==0) erro("Erro"); a=a/255.0;
  Mat_<float> q=imread("letramore.pgm",0); if (q.total()==0) erro("Erro"); q=q/255.0;
  q=somaAbsDois( dcReject(q) );
  Mat_<float> p=filtro2d(a,q);
  imwrite("correlacao3.png",255*p);

  Mat_<Vec3f> d;
  cvtColor(a,d,COLOR_GRAY2BGR);
  for (int l=0; l<a.rows; l++)
    for (int c=0; c<a.cols; c++)
      if (p(l,c)>=0.64)
        rectangle(d,Point(c-109,l-38),Point(c+109,l+38),Scalar(0,0,1),3);
  imwrite("ocorrencia3.png",255*d);
}
```

Programa P3: Fazendo template matching habirual, não consegue achar palavras "mo?e" onde a terceira letra foi alterada.

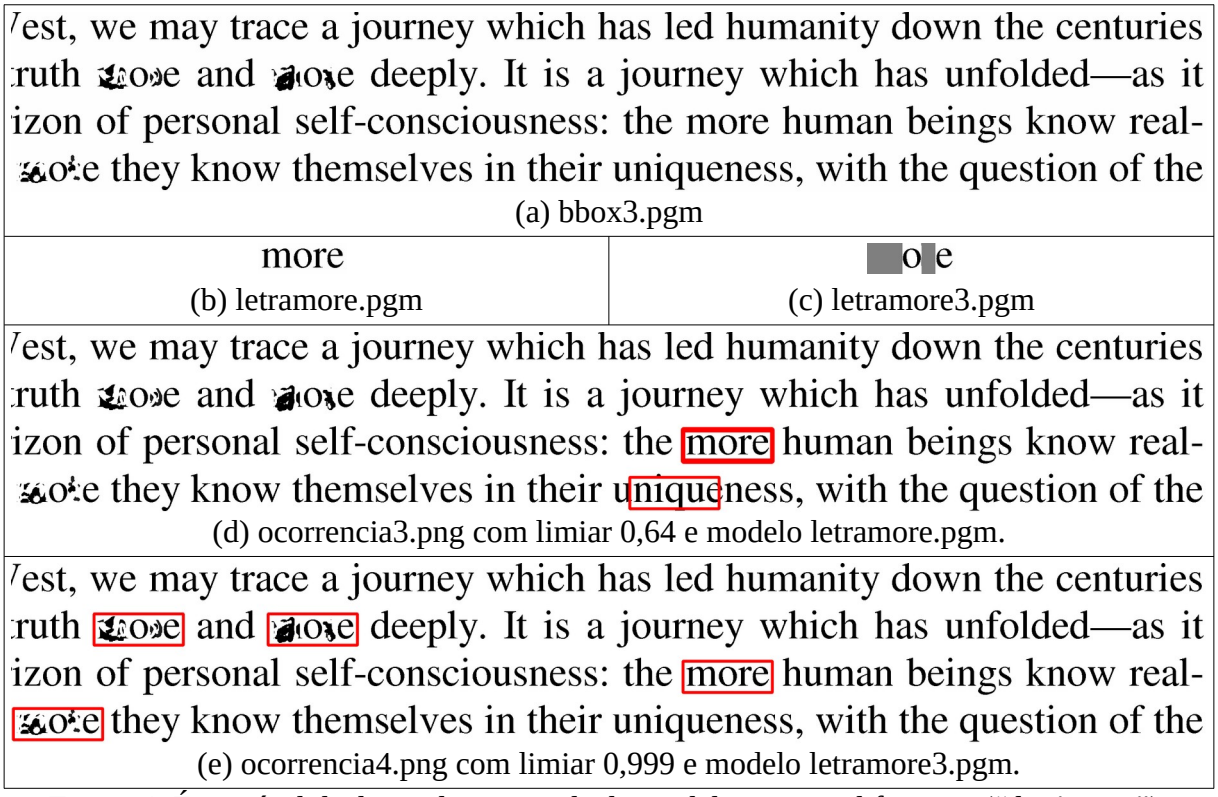


Figura 4: É possível declarar alguns pixels do modelo como indiferentes ("don't care").

Repare que somente a primeira e a terceira letras da palavras “more” foram danificadas, enquanto que a segunda e a quarta letras permaneceram sem deformação. Neste caso, podemos informar ao template matching que não leve em consideração primeira e terceira letras, procurando pelo padrão “?o?e”. Lembre-se de que estamos calculando média aritmética ponderada para efetuar casamento de modelo. Para que o valor de algum pixel não interfira no cálculo da média ponderada, basta colocar peso zero nesse pixel.

Nota: A soma de todos os pixels do modelo deve continuar resultando zero e a soma absoluta de todos os pixels não-nulos do modelo deve continuar resultando em dois, para que casamento de modelo funcione corretamente.

A função *dcReject* chamada com 2 parâmetros consegue tratar modelos com pixels “don’t care”. O segundo parâmetro indica o nível de cinza dos pixels que devem ser considerados como indiferentes. Por exemplo, a chamada abaixo considera pixels cinza (128) como indiferentes, mapeando-os em zero.

```
dcReject( q, 128.0/255.0 )
```

O programa 2 conseguiu detectar todas as 4 palavras (figura 4e), usando limiar 0,999.

```
//match2d4.cpp 2024
#include "procimagem.h"
int main() {
  Mat_<float> a=imread("bbox3.pgm",0); a=a/255.0;
  Mat_<float> q=imread("letramore3.pgm",0); q=q/255.0;
  q=somaAbsDois( dcReject(q, 128.0/255.0) );
  Mat_<float> p=filtro2d(a,q);
  imwrite("correlacao4.png",255*p);

  Mat_<Vec3f> d;
  cvtColor(a,d,COLOR_GRAY2BGR);
  for (int l=0; l<a.rows; l++)
    for (int c=0; c<a.cols; c++)
      if (p(l,c)>=0.999)
        rectangle(d,Point(c-109,l-38),Point(c+109,l+38),Scalar(0,0,1),3);
  imwrite("ocorrencia4.png",255*d);
}
```

Programa 2: Usando “don’t care”, é possível achar palavras “?o?e”.

Exercício: Modifique o programa de *maematch.cpp* que você excreveu no exercício anterior, para que possa fornecer modelos com pixels “don’t care”.

5. Brilho e Contraste

Vamos definir que duas imagens $\boldsymbol{x}$ e $\boldsymbol{y}$ são equivalentes sob variação de brilho e contraste se existirem fatores de correção de contraste $\beta \neq 0$ e de brilho γ tais que $\boldsymbol{y} = \beta \boldsymbol{x} + \gamma \mathbf{1}$, onde $\mathbf{1}$ é a matriz de 1's. Em linguagem de Eletricidade, contraste seria "gain" e brilho seria "offset" ou "bias".

Na equação acima, se aumentar contraste colocando por exemplo $\beta=2$, um pixel cinza claro vai se tornar muito mais claro enquanto que cinza escuro vai se tornar um pouco mais claro. Não é isto que entendemos usualmente pelo aumento de contraste – aumentando contraste, cinza claro precisa ficar mais claro e cinza escuro precisa ficar mais escuro. A equação $\boldsymbol{y} = (\beta(\boldsymbol{x} - 0.5) + 0.5) + \gamma \mathbf{1}$ faz correção de brilho/contraste de forma mais intuitiva (supondo que o nível de cinza dos pixels de $\boldsymbol{x}$ e $\boldsymbol{y}$ vão de 0 a 1).

Figura: (a) Lenna original. (b) Lenna com aumento de brilho [p = p+γ]. (c) Lenna com aumento de contraste [p = (β(p -0.5))+0.5].

Invariante ao brilho e proporcional a contraste do sinal A

A saída do casamento de modelo visto acima é invariante ao brilho e proporcional a contraste. Para entender o que isto significa, vamos considerar um outro exemplo unidimensional.

```
//match1d2.cpp 2024
#include "procimagem.h"
int main() {
  Mat_<float> a = ( Mat_<float>(1,13) <<
                  0, 1, 5, 3, 1, -1, 3, 1, 1, -2, 6, 2, 0); cout << a << endl;
  Mat_<float> q = ( Mat_<float>(1,3) << 0, 1, 0.5 ); cout << q << endl;

  Mat_<float> q2=somaAbsDois(dcReject(q)); cout << q2 << endl;
  Mat_<float> p2=filtro2d(a,q2,BORDER_REPLICATE); cout << p2 << endl;
}
```

Programa 3: O casamento de modelo usando filtro linear é invariante a brilho e proporcional a contraste.

A saída obtida executando o programa 3 é:

```
a =  [0, 1, 5,  3,  1, -1, 3,   1, 1, -2, 6, 2,  0]
q =  [0, 1, 0.5]
q2 = [-1, 1, 0]
p2 = [0, 1, 4, -2, -2, -2, 4, -2, 0,  -3, 8, -4, -2]
```

No vetor *a*, as instâncias equivalentes (sob variação de brilho e contraste) ao modelo *q* aparecem três vezes e estão destacadas em amarelo.

```
1) [ 1, 5, 3] = 4*[0, 1, 0.5]+1
2) [-1, 3, 1] = 4*[0, 1, 0.5]-1
3) [-2, 6, 2] = 8*[0, 1, 0.5]-2
```

As respectivas saídas no vetor *p2* são 4, 4 e 8. Os brilhos das instâncias ("offsets" +1, -1, -2) não influenciaram nas saídas. Por outro lado, as saídas foram proporcionais aos contrastes ("gain": 4, 4, 8).

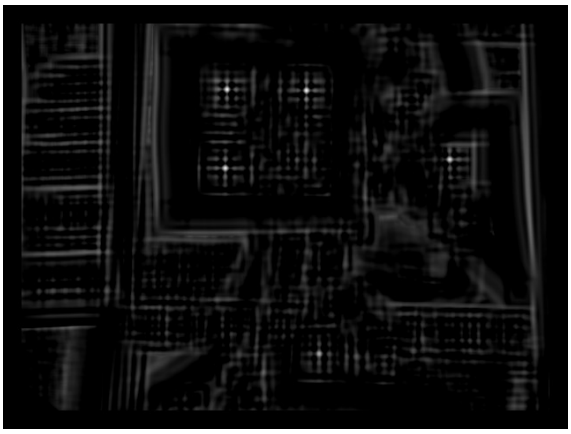
Proporcional a contraste da imagem 2D "a analisar" A

Esta propriedade, aplicado num exemplo 2D, está ilustrada na figura 5. A imagem da figura 5a foi editada manualmente para que as 5 instâncias do modelo (figura 5b) tivessem contrastes diferentes. O modelo da figura 5b foi redimensionada manualmente para ficar com tamanho semelhante às instâncias que aparecem na figura 5a. A saída 5c mostra a saída do filtro linear, com alturas das correlações (níveis de cinza dos 5 pontos brilhantes) nas 5 instâncias proporcionais aos contrastes das instâncias.

(a) op00.jpg com cinco instâncias com diferentes contrastes.

(b) padrao_reduz.png

(c) qr-p2.png gerado por filtro linear ou correlação cruzada TM_CCORR (CC)



(d) qr-p3.png gerado por coeficiente de correlação normalizada TM_CCOEFF_NORMED (NCC).

(e)

(f)

Figura 5: Casamento de modelo. (c, e) Casamento de modelo usando filtro linear ou correlação cruzada (CC) é invariante a brilho mas proporcional ao contraste. (d, f) Casamento de modelo usando coeficiente de correlação normalizada (NCC) é invariante a brilho e contraste.

6. Função *matchTemplate*

6.1 Correlação cruzada (CC) e coeficiente de correlação normalizada (NCC)

Até agora, usamos a função *filter2D* (ou *filtro2d*) para efetuar casamento de modelo. Porém, OpenCV possui uma função própria para fazer casamento de modelo: *matchTemplate*. Esta função sempre trabalha no modo "valid", isto é, a imagem de saída *P* é menor que a entrada *A*. Podemos imaginar que o modelo *Q* encontrado em *A* é indicado por um pico de correlação no canto superior esquerdo do modelo no resultado *P* (figura 6).

Nota: A função *matchTemplate* é altamente otimizada. Utiliza FFT (Fast Fourier Transform) e outros métodos para acelerar o processamento.

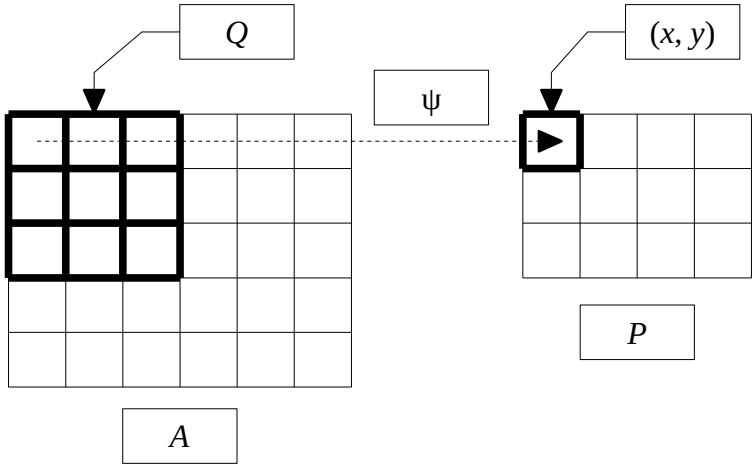


Figura 6: Casamento de modelo percorre a imagem *A* comparando template *Q* com cada posição (*x*, *y*) de *A*. O resultado da comparação é armazenado em resultado *P*(*x*,*y*). A saída *P* é menor que a entrada *A*.

A sintaxe do matchTemplate é:

```
C++: void matchTemplate(InputArray image, InputArray templ, OutputArray result, int method)
Ex: matchTemplate(A,Q,P,TM_CCORR)
Ex: matchTemplate(A,Q,P,TM_CCOEFF_NORMED)

Python: cv2.matchTemplate(image, templ, method[, result ]) → result
Ex: P=cv2.matchTemplate(A,Q,cv2.TM_CCORR)
Ex: P=cv2.matchTemplate(A,Q,cv2.TM_CCOEFF_NORMED)
```

Implementei a seguinte função em *procimagem.h* que faz *matchTemplate* trabalhar no modo "same", fazendo a imagem de saída ter o mesmo tamanho da entrada e colocando o resultado no centro do modelo (em vez de canto superior esquerdo):

```
Mat_<float> matchTemplateSame(Mat_<float> a, Mat_<float> q, int method, float backg=0.0);
Ex: P=matchTemplateSame(A,Q,TM_CCORR,0.0)  = CC
Ex: P=matchTemplateSame(A,Q,TM_CCOEFF_NORMED,0.0)  = NCC
```

Aqui, *backg* é o valor com que a função irá preencher os pixels fora do domínio da imagem de saída da função *matchTemplate*, para que a imagem de saída tenha o mesmo tamanho que a de entrada.

A função *matchTemplate* (assim como *matchTemplateSame*) fornece seis métodos diferentes de comparação. Desses, vamos estudar apenas dois que são os mais úteis:

1) Método TM_CCORR (correlação cruzada ou ***CC***):

$$P(x,y)=\sum_{x',y'}\left[A(x',y')\cdot Q(x+x',y+y')\right]$$

Este é o método que estudamos até agora. Usa correlação cruzada (ou filtro linear).

Nota: Como vimos, antes de chamar este método, deve pré-processar o modelo *Q*:

```
P=somaAbsDois(dcReject(Q))
```

Sem este pré-processamento, este método não funciona. Este método é invariante por brilho e proporcional ao contraste.

2) Método TM_CCOEFF_NORMED (coeficiente de correlação normalizada ou ***NCC***):

$$P(x,y)=\frac{\sum_{x'}\sum_{y'}\left[\widetilde{A}(x',y')\cdot\widetilde{Q}(x+x',y+y')\right]}{\sqrt{\sum_{x'}\sum_{y'}\left[\widetilde{A}(x',y')\right]^2\sum_{x'}\sum_{y'}\left[\widetilde{Q}(x+x',y+y')\right]^2}}$$

onde $\widetilde{Q}$ é template *Q* com correção de média e $\widetilde{A}$ é o conteúdo da imagem *A* dentro da janela móvel com correção de média.

Neste método, não é necessário fazer qualquer pré-processamento, pois a própria função *matchTemplate* se encarregará de fazer todos os pré-processamentos necessários.

Nota: Repare que o numerador é o mesmo do método CC.

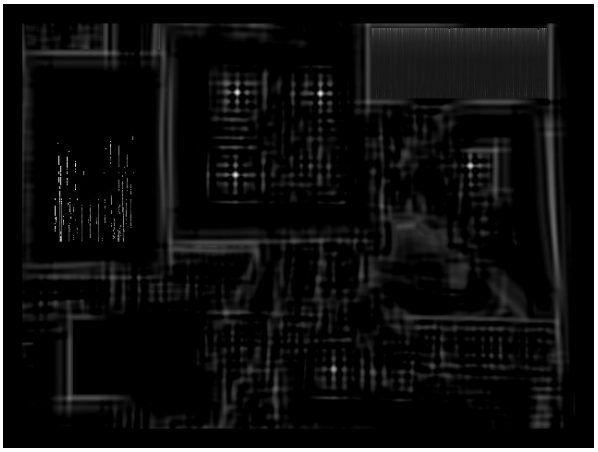
No método NCC, deve-se tomar cuidado com possíveis divisões por zero. A divisão por zero ocorre se o conteúdo de *A* dentro da janela tiver nível de cinza constante. No caso de divisão por zero, a saída de NCC fica indeterminada (figura X). Este método é invariante por brilho e contraste.

Nota: A divisão por zero também ocorre se todos os pixels de *Q* tiverem o mesmo valor de cinza. Porém, na prática não precisamos nos preocupar com este caso, pois ninguém vai querer buscar um modelo com todos os pixels iguais.

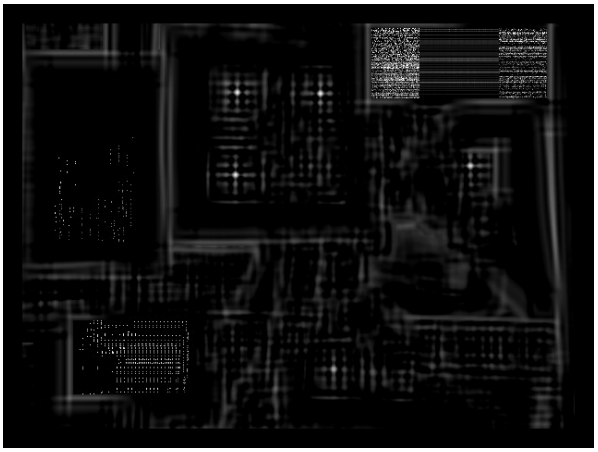
Nota: O artigo [Lewis1995] explica como é possível calcular NCC rapidamente usando FFT e outras técnicas.

A=op00.png

Q=padrao_reduz.png



divisao0.png (saída de OpenCV2)



divisao0-v3.png (saída de OpenCV3)

Figura X: Os valores das saídas geradas pela função *matchTemplate* de OpenCV2 e OpenCV3 nas regiões com nível de cinza constante são indeterminadas.

6.2 Notação de vetor

Uma outra forma de descrever os dois métodos, para facilitar a compreensão, é usar a notação de vetor. Sejam dados vetores x e y (x é o conteúdo dentro da janela da imagem A e y é o conteúdo da imagem Q escrito em forma de vetor). Vamos denotar esses dois vetores após correção de média (*dcReject*) de $\tilde{x}$ e $\tilde{y}$. Isto é, $\tilde{x}=x-\bar{x}$, onde $\bar{x}$ é a média de x (o mesmo vale para y).

1) Método CV_TM_CCORR (correlação cruzada ou CC).
Este método calcula o produto escalar dos dois vetores.

$$p=\sum_i x_i \cdot y_i = x \cdot y$$

onde · indica o produto escalar. Fazendo "dcReject" como pré-processamento, temos:

$$p=\tilde{x}\cdot\tilde{y}=x\cdot\tilde{y}$$

2) Método CV_TM_CCOEFF_NORMED (coeficiente de correlação normalizada ou NCC).
Este método calcula o cosseno do ângulo formado pelos dois vetores com correção de média.

$$p=\frac{\tilde{x}\cdot\tilde{y}}{\|\tilde{x}\|\|\tilde{y}\|}=\cos(\theta)$$

onde θ é o ângulo formado pelos dois vetores. Este ângulo é o coeficiente de correlação (de Pearson) e é amplamente usado em Estatística para quantificar correlação linear entre duas variáveis.

6.X Exemplo 1D de template matching NCC

```
1  //match1d3.cpp 2024
2  #include "procimagem.h"
3  int main() {
4    Mat_<float> a = ( Mat_<float>(1,13) <<
5                    0, 1, 5, 3, 1, -1, 3, 1, 1, -2, 6, 2, 0); cout << a << endl;
6    Mat_<float> q = ( Mat_<float>(1,3) << 0, 1, 0.5 ); cout << q << endl;
7
8    Mat_<float> q2=somaAbsDois(dcReject(q)); cout << q2 << endl;
9    Mat_<float> p2=matchTemplateSame(a,q2,TM_CCORR); cout << p2 << endl;
10
11   Mat_<float> p3=matchTemplateSame(a,q,TM_CCOEFF_NORMED); cout << p3 << endl;
12 }
```

Programa P: A saída do casamento de modelo usando filtro linear é invariante a brilho e proporcional a contraste.

```
a =  [0, 1, 5, 3, 1, -1, 3, 1, 1, -2, 6, 2, 0]
q =  [0, 1, 0.5]
q2 = [-1, 1, 0]
p2 = [0, 1, 4, -2, -2, -2, 4, -2, 0, -3, 8, -4, 0]   (CC)
p3 = [0, 0.189, 1, -0.5, -0.5, -0.5, 1, -0.866, 0, -0.371, 1, -0.655, 0]   (NCC)
```

O programa P mostra a busca do template *q* no vetor *a* usando template matchings CC (*p2*) e NCC (*p3*). A saída de CC é igual à de filtro linear (programa 3), gerando saídas proporcionais aos contrastes. Enquanto isso, a saída de NCC gerou valor exatamente 1 nas três instâncias, independentemente do contraste.

6.3 Exemplo QR-code

Figura 5 mostra um exemplo de casamento de modelos CC e NCC. As saídas 5c e 5e foram geradas usando filtro linear (CC) e os 5 picos de correlação possuem brilhos proporcionais aos contrastes das instâncias. As saídas 5d e 5f foram geradas usando *matchTemplate* com método correlação cruzada normalizada (NCC). Os 5 picos possuem mais ou menos o mesmo brilho pois este método é invariante a brilho e contraste. O programa usado para gerar a figura 5 está no programa 4 abaixo.

```
1  //qr.cpp - 2024
2  #include "procimagem.h"
3
4  Mat_<Vec3f> marca(Mat_<float> a, Mat_<float> p, float limiar) {
5    Mat_<Vec3f> d;
6    cvtColor(a,d,COLOR_GRAY2BGR);
7    for (int l=0; l<a.rows; l++)
8      for (int c=0; c<a.cols; c++)
9        if (p(l,c)>=limiar)
10         rectangle(d,Point(c-25,l-25),Point(c+25,l+25),Scalar(0.0,0.0,1.0),3);
11   return d;
12 }
13
14 int main() {
15   Mat_<float> a=imread("op00.jpg",0);
16   a = a / 255.0;
17   Mat_<float> q=imread("padrao_reduz.png",0);
18   q = q / 255.0;
19
20   Mat_<float> q1=somaAbsDois(dcReject(q));
21   Mat_<float> p1=filtro2d(a,q1);
22   imwrite("qr-p1.png",255.0*p1);
23   Mat_<Vec3f> m1=marca(a,p1,0.6);
24   imwrite("qr-m1.png",255.0*m1);
25
26   Mat_<float> p2=matchTemplateSame(a,q1,TM_CCORR);
27   imwrite("qr-p2.png",255.0*p2);
28   Mat_<Vec3f> m2=marca(a,p2,0.6);
29   imwrite("qr-m2.png",255.0*m2);
30
31   Mat_<float> p3=matchTemplateSame(a,q,TM_CCOEFF_NORMED);
32   imwrite("qr-p3.png",255.0*p3);
33   Mat_<Vec3f> m3=marca(a,p3,0.6);
34   imwrite("qr-m3.png",255.0*m3);
35 }
```

Programa 4: Exemplo de matchTemplateSame.

Nota: As saídas geradas pela *filtro2d* e pela *matchTemplateSame* (modo CC) são praticamente iguais (exceto nas bordas das imagens de saída p1 e p2).

Exercício: Escreva um programa que localiza em op00.jpg a instância de padrao_reduz.png com o maior contraste.

op00.jpg

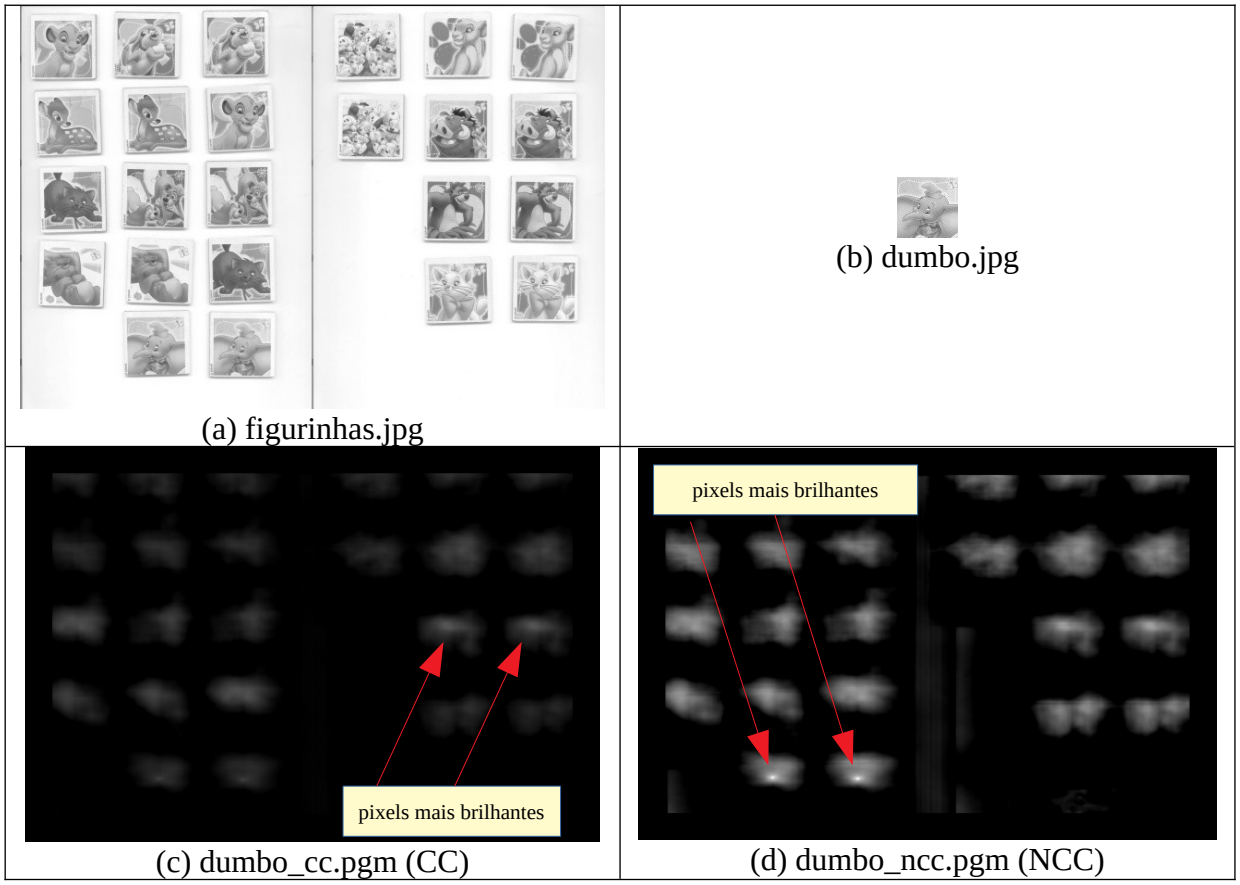
padrao_reduz.png

6.4 Exemplo Dumbo

Mais um exemplo para ilustrar diferença entre CC e NCC. O programa 7 abaixo faz busca do modelo *Q* “dumbo.jpg” na imagem *A* “figurinhas.jpg” usando CC e NCC (figura 7). Os dois pixels mais brilhantes do casamento de modelo usando CC não são as localizações do elefante “Dumbo”, mas do urso “Baloo”. Repare que “Dumbo” tem baixo contraste e “Baloo” tem alto contraste. CC gera correlação alta onde tem alto contraste, mesmo que a instância não case perfeitamente com o modelo. NCC, por outro lado, consegue indicar corretamente as duas ocorrências de “Dumbo”, pois é invariante a contraste.

```
1  // dumbo.cpp - grad2020
2  #include "procimagem.h"
3  int main() {
4    Mat_<float> a=imread("figurinhas.jpg",0); a=a/255;
5    Mat_<float> q=imread("dumbo.jpg",0); q=q/255;
6    Mat_<float> q2=somaAbsDois(dcReject(q));
7    Mat_<float> p1=matchTemplateSame(a, q2, TM_CCORR);
8    imwrite("dumbo_cc.pgm",255*p1);
9    Mat_<float> p2=matchTemplateSame(a, q, TM_CCOEFF_NORMED);
10   imwrite("dumbo_ncc.pgm",255*p2);
11 }
```

Programa 7: Exemplo 2D para verificar diferença entre CC e NCC.



(a) figurinhas.jpg (b) dumbo.jpg (c) dumbo_cc.pgm (CC) (d) dumbo_ncc.pgm (NCC)

Figura 7: Procurando “Dumbo” com template matching CC, obtemos localização de “Baloo”, pois “Dumbo” tem baixo contraste e “Baloo” tem alto contraste. Procurando “Dumbo” com NCC, obtemos localização correta, apesar do baixo contraste de “Dumbo”.

[PSI3471 – Pular a partir daqui.]

6.5 Explicação

A figura 8 tenta explicar graficamente o que está acontecendo, representando os valores dos pixels do "modelo Dumbo", "verdadeira ocorrência de Dumbo" e "falsa ocorrência de Dumbo" como 3 vetores. O vetor preto de comprimento 10 que aparece nas figuras 8a e 8b representa os níveis de cinza do "modelo Dumbo" (estamos usando dimensão 2 para poder desenhar no papel, mas o vetor teria número de dimensões igual a número de pixels do modelo).

Vamos supor que o "Dumbo" aparece com contraste menor na imagem analisada *A* do que no modelo *Q*, representado pelo vetor vermelho da figura 8a de comprimento 5. Na figura 8a, os dois vetores apontam para a mesma direção, pois o modelo Dumbo e ocorrência Dumbo representam a mesma figura (com contrastes diferentes). Calculando o produto escalar (CC) dá 50. Calculando o cosseno do ângulo entre os vetores (NCC) dá 1.

Por outro lado, "Baloo" aparece com alto contraste na imagem de busca, representado pelo vetor vermelho da figura 8b de comprimento 14. Na Figura 8b, os dois vetores não apontam para a mesma direção, pois Dumbo e Baloo não representam a mesma figura. Calculando o produto escalar (CC) dá 100. Calculando o cosseno do ângulo entre os vetores (NCC) dá 0,7.

Repare que o produto escalar da instância "Baloo" (figura 8b, 100) deu maior do que o produto escalar da instância "Dumbo" (figura 8a, 50), pois "Baloo" tem contraste bem mais alto do que Dumbo.

Por outro lado, o cosseno de ângulo da instância "Dumbo" (figura 8a, 1) dá maior do que o cosseno de ângulo da instância "Baloo" (figura 8b, 0.7), pois o cosseno do ângulo mede se os dois vetores representam a mesma figura (isto é, se estão apontando para a mesma direção).

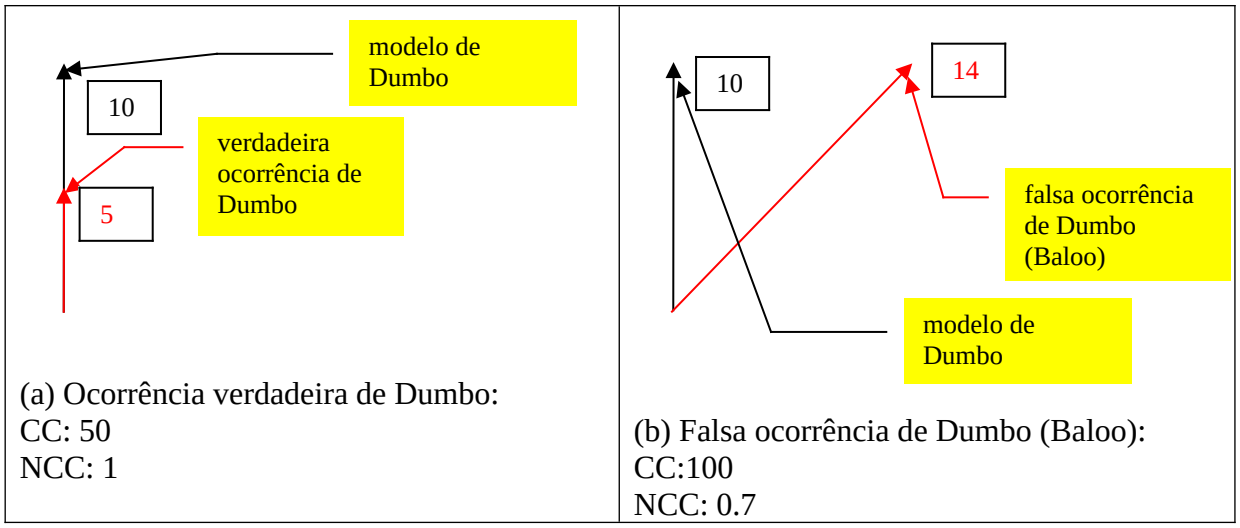


Figura 8: Uma explicação gráfica de por que CC falhou ao localizar Dumbo.

[PSI3471 – Pular até aqui.]

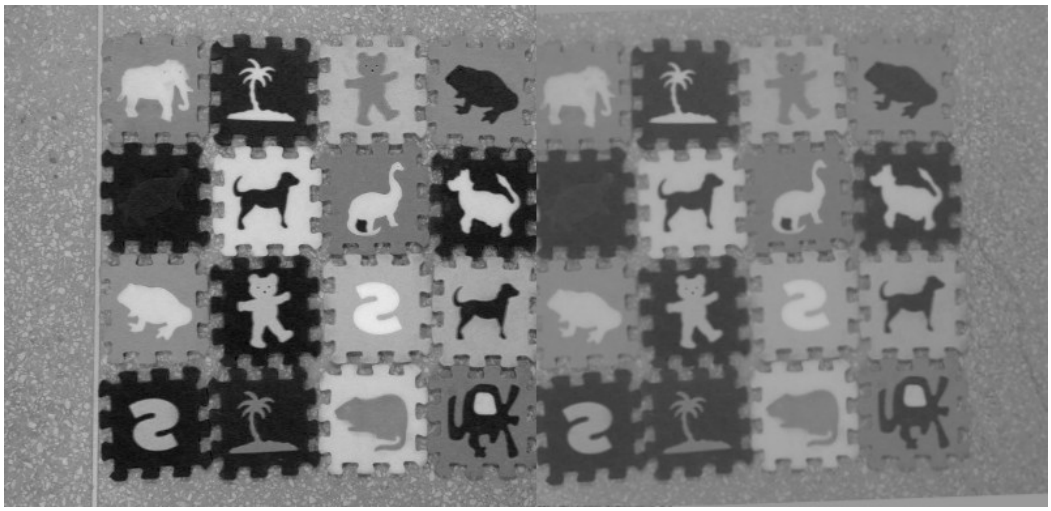
Exercício: Descreva numa aplicação onde é melhor usar CC do que NCC.
Resposta: Quando queremos localizar um modelo que com certeza aparece com alto contraste em *A*. Por exemplo, a figura 1 desta apostila.

Exercício: Como podemos detectar um modelo *Q*, se este pode aparecer em diferentes escalas (tamanhos) na imagem a analisar *A*?

Exercício: Como podemos detectar um modelo *Q*, se este pode aparecer em diferentes ângulos (rotações) na imagem a analisar *A*?

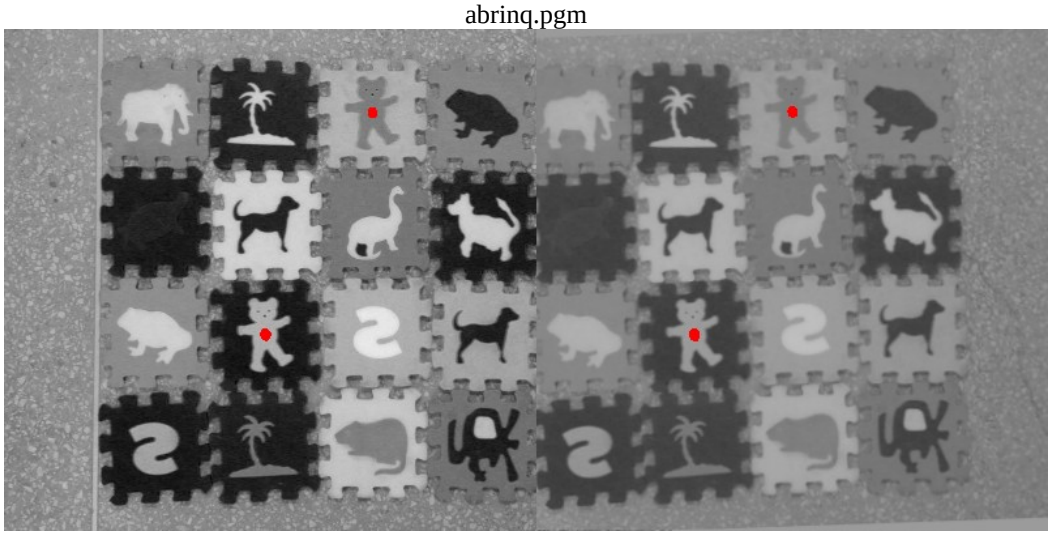
[PSI3471 aula 5/6 – Lição de casa #1/1. Vale 10.] Escreva um programa usando função *matchTemplate* que detecta as 4 ocorrências de urso "q.png" na imagem a analisar "a.png" gerando uma imagem processada semelhante a "p.png", chamando uma única vez a função *matchTemplate* e sem detectar previamente as bordas.

Nota: Este exercício pode ser resolvido usando template matching aplicado diretamente em níveis de cinza. Não detecte as bordas antes de chamar template matching. Detectar bordas é uma solução muito menos robusta do que fazer template matching direto. Pense o seguinte. Digamos que você tenha detectado as bordas das imagens a e q abaixo e tenha conseguido um casamento das bordas das duas imagens. Neste caso, as bordas de a e q de deixam de "bater" se deslocar q em um pixel em qualquer direção, se mudar minimamente a escala ou fizer uma pequena rotação. Enquanto isso, o casamento levando em conta o nível de cinza das imagens continuam "batendo" mesmo com pequenas distorções.



abrinq.pgm

qbrinq.pgm



pbrinq.ppm

Solução: ~/algpi/tmatch/grad2025/urso.cpp

[PSI3471 aulas 5/6. Lição de casa extra. Vale +4 pontos.]

Cada uma das 12 imagens *q??.jpg* aparece uma única vez na imagem *a.jpg*, possivelmente rotacionado. Faça um programa que lê as imagens *a.jpg* e as 12 imagens-modelos *q??.jpg* e gera a imagem *p.jpg* indicando onde está cada uma das 12 imagens-modelos juntamente com o ângulo da rotação, como na figura abaixo à direita.

Sugestão: Você pode rotacionar as imagens *q??.jpg* em vários ângulos e buscá-los todos na imagem *a.jpg*.

Imagem a.jpg

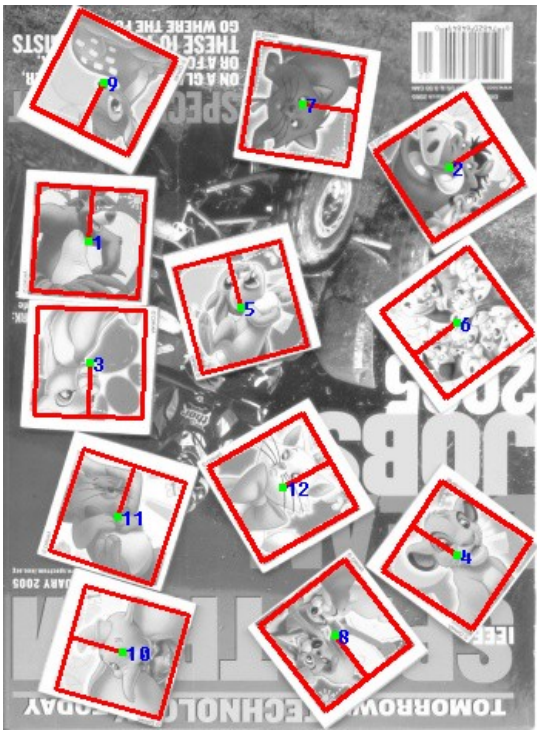


Imagem de saída p.jpg procurando todas as imagens q??.jpg.

Imagens q??.JPG

Se você baixou as imagens conforme:

```
$ wget -U 'Firefox/50.0' http://www.lps.usp.br/hae/apostila/tmatch.zip
$ unzip tmatch.zip
```

as imagens a serem usadas estão no subdiretório tmatch_extra.

A função *rotacao* abaixo rotaciona uma imagem em níveis de cinza *ent* por um ângulo *graus* em torno do ponto *centro* e devolve a imagem resultante *sai* com tamanho *tamanho*. Estudaremos este tópico com mais detalhes na próxima aula.

```
#include "procimagem.h"

Mat_<uchar> rotacao(Mat_<uchar> ent, double graus, Point2f centro, Size tamanho) {
  Mat_<double> m=getRotationMatrix2D(centro, graus, 1.0);
  Mat_<uchar> sai;
  warpAffine(ent, sai, m, tamanho, INTER_LINEAR, BORDER_CONSTANT, Scalar(255));
  return sai;
}

int main() {
  Mat_<uchar> ent=imread("a.png",0);
  Mat_<uchar> sai=rotacao(ent,30,Point2f(ent.size())/2,ent.size());
  imwrite("rotacao.png",sai);
}
```

Referências:

[Lewis1995] J.P. Lewis, “Fast normalized cross-correlation,” *Vision Interface*, pp. 120-123, 1995, url = "citeseer.ist.psu.edu/lewis95fast.html"

[PSI3471 aula 5 - Fim]