VISÃO COMPUTACIONAL

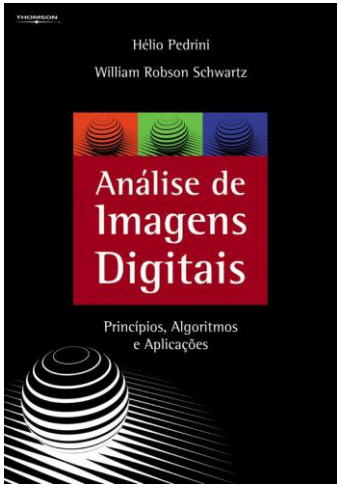
TRANSFORMAÇÕES GEOMÉTRICAS

Prof. Msc. Giovanni Lucca França da Silva
E-mail: giovanni-lucca@live.com

SOBRE A DISCIPLINA

- Bibliografia principal:
  - GONZALEZ, Rafael C.; WOODS, Richard C. **Processamento digital de imagens**. Pearson, 2011.

- Bibliografia complementar:
  - PEDRINI, Hélio; SCHWARTZ, William Robson. **Análise de imagens digitais: princípios, algoritmos e aplicações**. Thomson Learning, 2008.



NA AULA PASSADA...

- Conectividade.
- Componentes conexos.
- Operações lógico-aritméticas.

ROTEIRO

- Introdução.
- Ampliação e redução.
- Translação.
- Escala.
- Rotação.
- Espelhamento.

INTRODUÇÃO

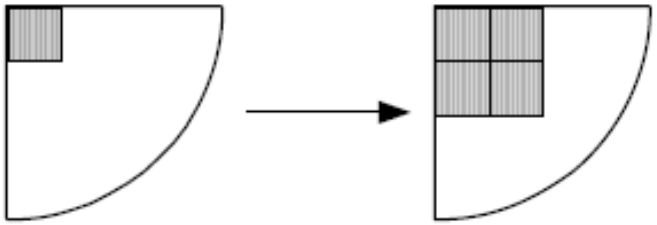
- Operações de processamento de imagens cujo principal efeito é a alteração da posição espacial dos pixels que a compõem.

- Aplicações:
  - Correção de distorções.
  - Aumento de base de dados.
  - Produção de efeito artístico.

AMPLIAÇÃO E REDUÇÃO

- Em inglês, *zoom in* e *zoom out*, são processos pelos quais as dimensões de uma imagem são aumentadas ou diminuídas para efeito de visualização.

- A maneira mais simples de ampliar uma imagem é duplicar os valores dos pixels na direção X ou Y ou em ambas.



Expansão de um pixel em 4 (*zoom* 2x)

AMPLIAÇÃO E REDUÇÃO

- Para reduzir as dimensões de uma imagem de um fator 2, basta utilizar o processo inverso, isto é converter cada agrupamento de quatro pixels novamente em 1 pixel.

- Problema: Poderá haver perda de informação no processo de *zoom out*.

- Solução: Média dos quatro pixels equivalentes na imagem original.

AMPLIAÇÃO E REDUÇÃO

TRANSLAÇÃO

- Consiste basicamente no deslocamento linear de cada pixel de coordenadas (X,Y,Z) na horizontal e/ou na vertical, mapeando para o ponto de coordenadas (X*,Y*,Z*).

$$X^* = X + X_0$$
$$Y^* = Y + Y_0$$
$$Z^* = Z + Z_0$$

$$\begin{bmatrix} X^* \\ Y^* \\ Z^* \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

TRANSLAÇÃO

- É sempre útil concatenar-se várias transformações para produzir um resultado composto.
  - Solução: uso de matrizes quadradas.

$$\begin{bmatrix} X^* \\ Y^* \\ Z^* \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} X^* \\ Y^* \\ Z^* \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

TRANSLAÇÃO

- Código.

C++: void **warpAffine**(InputArray **src**, OutputArray **dst**, InputArray **M**, Size **dsize**, int **flags**=INTER_LINEAR, int **borderMode**=BORDER_CONSTANT, const Scalar& **borderValue**=Scalar())

Python: `cv2.warpAffine`(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]]) → dst

Parameters:

- **src** – input image.
- **dst** – output image that has the size `dsize` and the same type as `src` .
- **M** – 2×3 transformation matrix.
- **dsize** – size of the output image.
- **flags** – combination of interpolation methods (see `resize()`) and the optional flag `WARP_INVERSE_MAP` that means that `M` is the inverse transformation (`dst` → `src`).
- **borderMode** – pixel extrapolation method (see `borderInterpolate()`); when `borderMode=BORDER_TRANSPARENT` , it means that the pixels in the destination image corresponding to the "outliers" in the source image are not modified by the function.
- **borderValue** – value used in case of a constant border; by default, it is 0.

TRANSLAÇÃO

- Código.

```
import numpy as np
import cv2 as cv

img = cv.imread('messi5.jpg',0)
rows,cols = img.shape

M = np.float32([[1,0,100],[0,1,50]])
dst = cv.warpAffine(img,M,(cols,rows))

cv.imshow('img',dst)
cv.waitKey(0)
cv.destroyAllWindows()
```

TRANSLAÇÃO

- Código.

```
int matrixTranslation[3][3];

for(int l = 0; l < 3; l++){
    for(int c = 0; c < 3; c++){
        if (l == c){ matrixTranslation[l][c] = 1;}
        else if (l == 0 && c == 2){ matrixTranslation[l][c] = x;}
        else if (l == 1 && c == 2){ matrixTranslation[l][c] = y;}
        else {matrixTranslation[l][c] = 0;}
    }
}
```

```
void Geometry::multMatrixTranslation(int matrix1[3][3], int matrix2[3][1], int matrix3[3][1]){

   int i, j, k;

   for(i = 0; i < 3; i++){
       for(j = 0; j < 1; j++){
           matrix3[i][j] = 0;
           for (k = 0; k < 3; k++){
               matrix3[i][j] += matrix1[i][k] * matrix2[k][j];
           }
       }
   }
}
```

TRANSLAÇÃO

- Código.

```
for(int i = 0; i < width; i++){
    for(int j = 0; j < height; j++){

        ImageType::IndexType pixelIndex={{i,j}};

        int pixel = image->GetPixel(pixelIndex);

        int matrixCoord[3][1] = {i,j,1};

        int matrixResult[3][1] = {x,y,1};

        multMatrixTranslation(matrixTranslation, matrixCoord, matrixResult);

        pixelIndex[0] = matrixResult[0][0];
        pixelIndex[1] = matrixResult[1][0];

        if(pixelIndex[0] >= width) pixelIndex[0] = width - 1;

        if(pixelIndex[1] >= height) pixelIndex[1] = height - 1;

        geometryImage->SetPixel(pixelIndex, pixel);

    }
}
```

ESCALA

- Refere-se ao caso em que a imagem é ampliada ou reduzida por um fator.
  - Conceito diferente do redimensionamento, porém tecnicamente idênticos.

$$\mathbf{S} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ESCALA

- Código.

C++: void **resize**(InputArray **src**, OutputArray **dst**, Size **dsize**, double **fx**=0, double **fy**=0, int **interpolation**=INTER_LINEAR)

Python: cv2.**resize**(src, dsize[, dst[, fx[, fy[, interpolation]]]]) → dst

Parameters:

- **src** – input image.
- **dst** – output image; it has the size `dsize` (when it is non-zero) or the size computed from `src.size()`, `fx`, and `fy`; the type of `dst` is the same as of `src`.
- **dsize** –

  output image size; if it equals zero, it is computed as:

  ```
  dsize = Size(round(fx*src.cols), round(fy*src.rows))
  ```

  Either `dsize` or both `fx` and `fy` must be non-zero.

ESCALA

- Código.

C++: void **resize**(InputArray **src**, OutputArray **dst**, Size **dsize**, double **fx**=0, double **fy**=0, int **interpolation**=INTER_LINEAR)

Python: cv2.**resize**(src, dsize[, dst[, fx[, fy[, interpolation]]]]) → dst

Parameters:

- **fx** –

  scale factor along the horizontal axis; when it equals 0, it is computed as

  `(double)dsize.width/src.cols`

- **fy** –

  scale factor along the vertical axis; when it equals 0, it is computed as

  `(double)dsize.height/src.rows`

ESCALA

- Código.

C++: void **resize**(InputArray **src**, OutputArray **dst**, Size **dsize**, double **fx**=0, double **fy**=0, int **interpolation**=INTER_LINEAR)

Python: cv2.**resize**(src, dsize[, dst[, fx[, fy[, interpolation]]]]) → dst

Parameters:

- **interpolation** –

  interpolation method:

  - **INTER_NEAREST** - a nearest-neighbor interpolation
  - **INTER_LINEAR** - a bilinear interpolation (used by default)
  - **INTER_AREA** - resampling using pixel area relation. It may be a preferred method for image decimation, as it gives moire'-free results. But when the image is zoomed, it is similar to the INTER_NEAREST method.
  - **INTER_CUBIC** - a bicubic interpolation over 4x4 pixel neighborhood
  - **INTER_LANCZOS4** - a Lanczos interpolation over 8x8 pixel neighborhood

ESCALA

- Código.

```
import numpy as np
import cv2 as cv

img = cv.imread('messi5.jpg')

res = cv.resize(img,None,fx=2, fy=2, interpolation = cv.INTER_CUBIC)

#OR

height, width = img.shape[:2]
res = cv.resize(img,(2*width, 2*height), interpolation = cv.INTER_CUBIC)
```

ESCALA

- Código.

```
int matrixScale[2][2];

for(int l = 0; l < 2; l++){
    for(int c = 0; c < 2; c++){
        if (l == 0 && c == 0){ matrixScale[l][c] = sx;}
        else if (l == 1 && c == 1){ matrixScale[l][c] = sy;}
        else { matrixScale[l][c] = 0;}
    }
}
```

```
void Geometry::multMatrixScale(int matrix1[2][2], int matrix2[2][1], int matrix3[2][1]){

    int i, j, k;

    for(i = 0; i < 2; i++){
        for(j = 0; j < 1; j++){
            matrix3[i][j] = 0;
            for (k = 0; k < 2; k++){
                matrix3[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

ESCALA

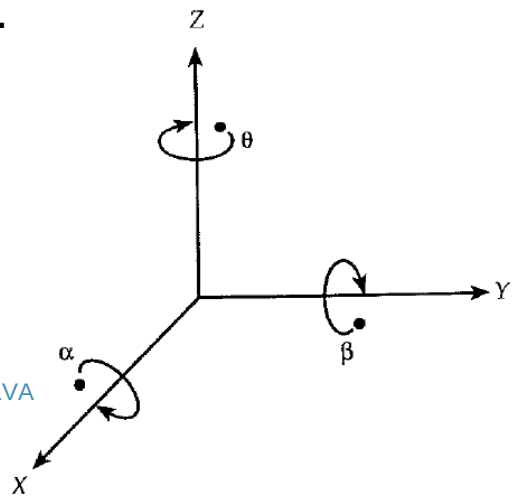
- Código.

```
for(int i = 0; i < width; i++){
    for(int j = 0; j < height; j++){

        ImageType::IndexType pixelIndex={{i,j}};

        int pixel = image->GetPixel(pixelIndex);

        int matrixCoord[2][1] = {i,j};

        int matrixResult[2][1] = {sx,sy};

        multMatrixScale(matrixScale, matrixCoord, matrixResult);

        pixelIndex[0] = matrixResult[0][0];
        pixelIndex[1] = matrixResult[1][0];

        if(pixelIndex[0] >= width) pixelIndex[0] = width - 1;
        if(pixelIndex[0] < 0) pixelIndex[0] = 0;

        if(pixelIndex[1] >= height) pixelIndex[1] = height - 1;
        if(pixelIndex[1] < 0) pixelIndex[1] = 0;

        geometryImage->SetPixel(pixelIndex, pixel);

    }
}
```

ROTAÇÃO

- Uma imagem pode ser rotacionada de um ângulo arbitrário, tanto no sentido horário quanto no anti-horário.

- A forma mais simples dessa operação é para rotacionar um ponto em torno dos eixos de coordenadas.



$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & \text{sen}\theta & 0 & 0 \\ -\text{sen}\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_\alpha = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \text{sen}\alpha & 0 \\ 0 & -\text{sen}\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_\beta = \begin{bmatrix} \cos\beta & 0 & -\text{sen}\beta & 0 \\ 0 & 1 & 0 & 0 \\ \text{sen}\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

ROTAÇÃO

- Três transformações são necessárias para rotacionar um ponto em torno de um outro ponto arbitrário no espaço.
  - Translada o ponto arbitrário para a origem.
  - Realiza a rotação.
  - Translada o ponto de volta para a sua posição original.

ROTAÇÃO

- Código.

C++: Mat **getRotationMatrix2D**(Point2f **center**, double **angle**, double **scale**)

Python: cv2.**getRotationMatrix2D**(center, angle, scale) → retval

Parameters:

- **center** – Center of the rotation in the source image.
- **angle** – Rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner).
- **scale** – Isotropic scale factor.

ROTAÇÃO

- Código.

```
img = cv.imread('messi5.jpg',0)
rows,cols = img.shape

# cols-1 and rows-1 are the coordinate limits.
M = cv.getRotationMatrix2D(((cols-1)/2.0,(rows-1)/2.0),90,1)
dst = cv.warpAffine(img,M,(cols,rows))
```

ROTAÇÃO

- Código.

```
float rad = (ang * 3.14)/180;

float matrixRotation[2][2];

matrixRotation[0][0] = cos(rad);
matrixRotation[0][1] = -1 * sin(rad);
matrixRotation[1][0] = sin(rad);
matrixRotation[1][1] = cos(rad);
```

```
void Geometry::multMatrixRotation(float matrix1[2][2], float matrix2[2][1], float matrix3[2][1]){

    int i, j, k;

    for(i = 0; i < 2; i++){
        for(j = 0; j < 1; j++){
            matrix3[i][j] = 0;
            for (k = 0; k < 2; k++){
                matrix3[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}
```

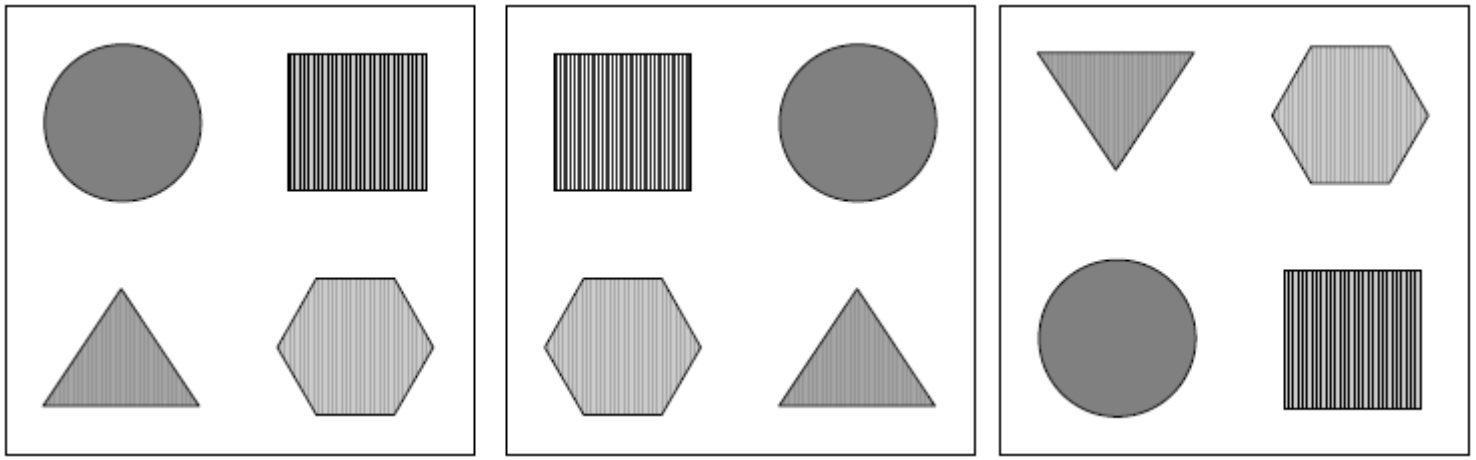
ROTAÇÃO

- Código.

```
for(int i = 0; i < width; i++){
    for(int j = 0; j < height; j++){

        ImageType::IndexType pixelIndex={{i,j}};

        int pixel = image->GetPixel(pixelIndex);

        float matrixCoord[2][1] = {i,j};

        float matrixResult[2][1];

        multMatrixRotation(matrixRotation, matrixCoord, matrixResult);

        pixelIndex[0] = round(matrixResult[0][0]);
        pixelIndex[1] = round(matrixResult[1][0]);

        if(pixelIndex[0] >= width) pixelIndex[0] = width - 1;
        if(pixelIndex[0] < 0) pixelIndex[0] = 0;

        if(pixelIndex[1] >= height) pixelIndex[1] = height - 1;
        if(pixelIndex[1] < 0) pixelIndex[1] = 0;

        geometryImage->SetPixel(pixelIndex, pixel);

    }
}
```

ESPELHAMENTO

- O espelhamento (*flip*) é uma operação que combina a rotação por ângulos múltiplos de 90º com o cálculo de matriz transposta.



Flip horizontal Flip vertical

ESPELHAMENTO

- Código.

C++: void **flip**(InputArray **src**, OutputArray **dst**, int **flipCode**)

Python: cv2.**flip**(src, flipCode[, dst]) → dst

Parameters:

- **src** – input array.
- **dst** – output array of the same size and type as src.
- **flipCode** – a flag to specify how to flip the array; 0 means flipping around the x-axis and positive value (for example, 1) means flipping around y-axis. Negative value (for example, -1) means flipping around both axes

ESPELHAMENTO

- Código.


```python
# import the OpenCV package
import cv2

# load the image with imread()
imageSource = 'images/messi5.jpg'
img = cv2.imread(imageSource)

# copy image to display all 4 variations
horizontal_img = img.copy()
vertical_img = img.copy()
both_img = img.copy()

# flip img horizontally, vertically,
# and both axes with flip()
horizontal_img = cv2.flip( img, 0 )
vertical_img = cv2.flip( img, 1 )
both_img = cv2.flip( img, -1 )

# display the images on screen with imshow()
cv2.imshow( "Original", img )
cv2.imshow( "Horizontal flip", horizontal_img )
cv2.imshow( "Vertical flip", vertical_img )
cv2.imshow( "Both flip", both_img )


# wait time in milliseconds
# this is required to show the image
# 0 = wait indefinitely
cv2.waitKey(0)

# close the windows
cv2.destroyAllWindows()
```

CONCATENAÇÃO DE OPERAÇÕES

- Rotação de uma imagem por um ângulo θ sobre um ponto x_c e y_c.

$$\begin{bmatrix} x_k \\ y_j \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_c \\ 0 & 1 & y_c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_c \\ 0 & 1 & -y_c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_q \\ v_p \\ 1 \end{bmatrix}$$

REFERÊNCIAS

- GONZALEZ, Rafael C.; WOODS, Richard C. **Processamento digital de imagens**. Pearson, 2011.

- PEDRINI, Hélio; SCHWARTZ, William Robson. **Análise de imagens digitais: princípios, algoritmos e aplicações**. Thomson Learning, 2008.

- SILVA, Aristófanes. **Notas de aula da disciplina Processamento de Imagens da Universidade Federal do Maranhão**. 2018.

- BRAZ Jr, Geraldo. **Notas de aula da disciplina Visão Computacional da Universidade Federal do Maranhão**. 2018.