

# Visão por Computador - Guião 3

Miguel Azevedo  
lobaoazevedo@ua.pt  
38569

Gabriel Vieira  
gabriel.vieira@ua.pt  
68021

## *Resumo –*

Este relatório descreve a resolução do terceiro guião prático da disciplina de Visão por Computador.

## *Palavras chave –*

Visão por computador, deteção de arestas, OpenCV, Canny, Laplacian, Sobel, Hough

## I. INTRODUÇÃO

Este guião prático consiste na deteção de arestas na imagem, usando as bibliotecas do OpenCV. Neste relatório, iremos explicar o que foi feito em cada exercício e comentar os seus resultados.

### A. Exercícios propostos:

- **Exercício 1:** Implementar um programa para capturar imagens da camera digital e explorar as funcionalidades da função Sobel para calcular o seu gradiente. Depois explorar os seus parâmetros, comentar os resultados e explorar também o algoritmo de Scharr.
- **Exercício 2:** Baseado no exercício anterior calcular também a Laplaciana de uma imagem. Explorar os parâmetros da função e comentar os seus resultados
- **Exercício 3:** Implementar um programa que capture imagens da camera digital e efectue a deteção de arestas com o algoritmo de Canny. Explorar os efeitos ao alterar os parâmetros e comentar os resultados obtidos.
- **Exercício 4:** Implementar um programa para capturar imagens da camera digital e efectuar a deteção de cantos usando o algoritmo de Harri (explorar a função *cornerHarris*). Para cada canto detetado, desenhar um circulo ou um quadrado na imagem, usando as funções do OpenCV.
- **Exercício 5:** Implementar um programa que detecte círculos e linhas de uma imagem. Como é sugerido no guião, em primeiro lugar calcular a imagem binária com as arestas presentes na cena, ajustando os parâmetros para obter uma melhor deteção. Explorar também o uso da função *findContours* para identificar os contornos correspondentes aos objectos de interesse.

- **Exercício 6:** Implementar uma solução diferente da anterior usando o *Hough Line Transform* e o *Hough Circle Transform*

## II. RESOLUÇÃO DOS EXERCÍCIOS PROPOSTOS:

### Exercício 1

Neste exercício é explicado o significado de cada parâmetro das funções *Sobel* e *Scharr*, bem como os resultados obtidos alterando os mesmos. Também é descrito o processo de resolução do exercício. Para começar, a função *Sobel* é declarada da seguinte forma: *Sobel(src image, output image, ddepth, x order, y order, kernel size, scale, delta, BORDER DEFAULT)* em que:

- **src image:** imagem de entrada, neste caso, será a imagem binarizada resultante da aplicação da Gaussiana de Blur para reduzir o ruído da imagem
- **output image:** imagem de saída
- **ddepth:** profundidade da imagem
- **x order:** ordem da derivada na direção X
- **y order:** ordem da derivada na direção Y
- **kernel size:** tamanho do kernel de Sobel estendido. Deve ser igual a 1,3,5 ou 7.
- **scale:** escala das derivadas computarizadas, por defeito não é aplicada.
- **delta:** valor de delta que é adicionado aos resultados prior para os armazenar na imagem de saída (*output image*).
- **border type:** método de extrapolação do pixel.

Em relação à função de Scharr é declarada da seguinte forma: *Scharr(src, dst, ddepth, dx, dy, scale, delta, border type* em que:

- **src image:** imagem de entrada, neste caso, será a imagem binarizada resultante da aplicação da Gaussiana de Blur para reduzir o ruído da imagem
- **output image:** imagem de saída
- **ddepth:** profundidade da imagem
- **dx:** ordem da derivada de X
- **dy:** ordem da derivada de Y
- **scale:** escala das derivadas computarizadas, por defeito não é aplicada.
- **delta:** valor de delta que é adicionado aos resultados prior para os armazenar na imagem de saída (*output image*).
- **border type:** método de extrapolação do pixel.

O objectivo deste exercício passa por calcular o gradiente da imagem de 1<sup>a</sup>ordem, e como vimos temos duas maneiras de fazer usando os dois algoritmos aqui abordados. Depois de analisarmos os parâmetros das funções, pode se então proceder ao processo da resolução do exercício. Em primeiro, a seleção entre a execução do algoritmo na imagem é feita pelo teclado, ao qual ao premirmos a tecla 's' usamos o algoritmo de Scharr e a tecla 'd' para o algoritmo de Sobel. Uma vez escolhida uma das opções, é necessário primeiro eliminar o ruído da imagem, pois assim o gradiente terá melhores resultados no final. Esta operação passa por usar a função gaussianBlur. Depois disto, a imagem resultante será convertida em tons de cinza, pois esta etapa é vital para que a imagem resultante seja mesmo um gradiente. Após a aplicação da função cvtColor na transformação da imagem de RGB para Gray, vai ser então calculado o gradiente em X e em Y usando as respectivas funções Sobel e Scharr, que são responsáveis por calcular a derivada das duas componentes. Depois esses valores de gradiente x e y vão ser calculados os seus valores absolutos e convertidos em 8 bits, através da função *convertScaleAbs(grad x, abs grad x)*. No fim de obtermos os valores absolutos dos gradientes x e y é altura de juntar as duas imagens, usando a função *addWeighted* que pega nas duas imagens, aplica um valor alfa para o gradiente x e beta para o gradiente y que vão ser responsáveis por definir a percentagem de gradiente de cada uma das componentes na imagem resultante, que é o último argumento da função. Por fim é mostrado então o resultado do gradiente, usando Sobel e usando Scharr nas duas imagens que se seguem:



Fig. 1 - Imagem original e gradiente usando Sobel

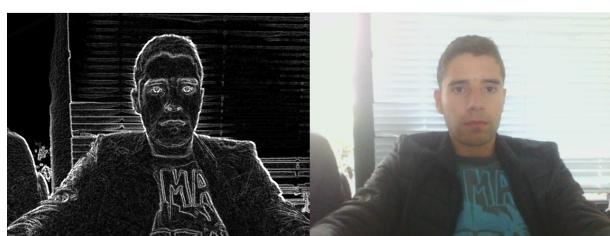


Fig. 2 - Imagem original e gradiente usando Scharr

Após a verificação das imagens, é possível concluir que aplicando o algoritmo de Scharr, obtemos uma melhor precisão na imagem do que usando Sobel.

A identificação baseia-se na diferença de valores entre pixels vizinhos, e a intensidade do branco é proporcional a essa diferença.

### Exercício 2

Pegando no exercício anterior, o objectivo deste era também calcular a Laplaciana da imagem original. Foi então adicionado uma nova opção à escolha no teclado. As operações a aplicar são iguais às aplicadas anteriormente às funções de Sobel e Scharr, mas aqui é aplicada a função *Laplacian* que tem os mesmos argumentos que a Sobel excepto as componentes x e y. É depois também calculado o valor absoluto da imagem de saída da função *Laplacian*, convertida em 8 bits e por fim mostrado o seu resultado, visível na figura seguinte:



Fig. 3 - Imagem original e sua Laplaciana

Olhando para o resultado da figura acima, podemos concluir que a Laplaciana é melhor para detectar arestas com mais precisão do que Sobel e Scharr.

### Exercício 3

Neste exercício, o objectivo passa por usar o algoritmo de *Canny* para detectar as arestas de uma dada imagem. Inicialmente é lida a imagem da camera digital e de seguida a imagem é convertida para tons de cinza (usando a função *cvtColor*) para depois ser mais fácil a deteção das arestas. Após a imagem estar em tons de cinza, é necessário usar a função *blur* (foi usada com kernel 3x3 mas pode ser ajustada para outros valores, dependendo do quanto se pretende reduzir o ruído da imagem) que reduz o ruído da imagem para que este não seja considerado como uma aresta da imagem. De seguida podemos aplicar o algoritmo de Canny sob a imagem filtrada. O a função de Canny é declarada da seguinte forma: *Canny(src image (gray scale), output image, threshold value, high threshold, kernel size)* em que:

- **src image:** imagem de entrada, neste caso, terá de estar em tons de cinza
- **output image:** imagem de saída
- **threshold:** valor de threshold aplicado à imagem, alterado pelo utilizador
- **high threshold:** valor máximo de threshold (3 vezes mais que o valor de threshold definido pelo utilizador, seguindo as recomendações de Canny)
- **kernel size:** tamanho do kerne, que neste caso

foi definido como 3 que é o kernel de Sobel a ser usado internamente.

Depois de usado o algoritmo de Canny, é possível ver o seu resultado, fazendo variar num trackbar o valor de threshold aplicado à imagem:

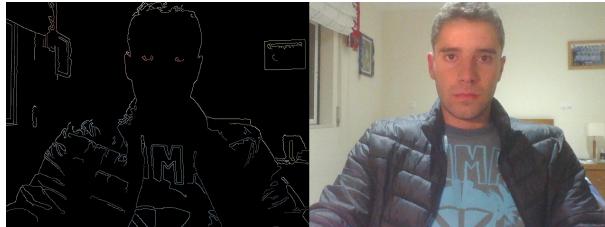


Fig. 4 - Imagem original e canny da mesma com threshold = 50

Com os resultados obtidos, observámos que à medida que fazímos variar o valor de threshold para perto de 50 que a deteção de arestas era mais precisa, enquanto que quanto menor fosse este valor a imagem tornava-se mais confusa porque a deteção era muito pouco precisa, e se o valor fosse perto de 100, poucas arestas restavam na imagem.

#### Exercício 4

Neste exercício, o objectivo era detetar os cantos de uma imagem e salientá-los com círculos. Para isso foi usado o algoritmo de *cornerHarris* que será explicado mais à frente. Inicialmente, a imagem é convertida em tons de cinza, porque assim, como foi dito para as arestas para os cantos é exactamente o mesmo, torna-se mais fácil identificá-los. Depois pegando na imagem em tons de cinza, aplicamos o algoritmo *cornerHarris* que faz a deteção de cantos na imagem. A função é declarada da seguinte forma: *cornerHarris(src, dst, blockSize, ksize, k, borderType)* em que:

- **src image:** imagem de entrada, neste caso, terá de estar em tons de cinza
- **output image:** imagem de saída
- **blockSize:** tamanho da vizinhança
- **ksize:** parâmetro de abertura para o operador de Sobel
- **k:** parâmetro livre do detector de arestas de Harris
- **borderType:** método de extrapolação do pixel.

Uma vez o algoritmo aplicado, é necessário normalizar a imagem resultante do algoritmo *cornerHarris*, definindo o valor máximo e mínimo, bem como fazer a conversão da imagem em 32 bits e por fim, antes de se desenhar os círculos, calcular o seu valor absoluto. A imagem abaixo mostra o resultado da aplicação do algoritmo:

Depois de analisar os resultados, podemos concluir que o algoritmo *cornerHarris* detecta os cantos de uma imagem com grande precisão. Também se



Fig. 5 - Imagem original e cornerHarris da mesma

pode concluir que se o tamanho dos blocos for muito elevado, mais difícil se torna encontrar os cantos da imagem, pois esta torna-se mais desfocada e confusa.

#### Exercício 5

Neste exercício foram usadas varias abordagens, para a determinação manual de círculos e linhas. Para os círculos a identificação de contornos circulares foi feita a procura dos pixéis de maior e menor valor X e Y e para poder ser estabelecido um raio e verificado se existiriam pixéis coincidentes com esse raio. Para as linhas foi usado um algoritmo que considera o primeiro ponto do contorno e calcula o declive da recta tangente aos dois pontos, comparando os declives obtidos em todas as iterações. Nenuma das abordagens funcionou, pelo que se pode concluir que "manualmente"é impossível fazer o que era pedido neste exercício.

#### Exercício 6

Neste exercício, o objectivo era implementar uma solução diferente da anterior, usando a *Hough Line Transform* e a *Hough Circle Transform*. Em ambos os casos, é necessário converter a imagem original para tons de cinza, sendo que na *Hough Line Transform* é usado antes o *Canny* para a deteção de arestas. Depois de obtermos a imagem em tons de cinza, é então aplicado a *Hough Circle Transform* e *Hough Line Transform*, no caso da primeira é aplicada sobre a imagem original, enquanto que na Hough Line é aplicada sobre a em tons de cinza resultante do canny, pois esta já tem as arestas detetadas para facilitar melhor a deteção das linhas. De referir que ainda no que toca à Hough Circle, antes de ser efectuada a respetiva operação é necessário reduzir o ruído da imagem, pois assim evita-se a falsa deteção de círculos na imagem. Para isto usou-se a função *gaussianBlur* (já explicada em trabalhos anteriores). A função *HoughCircles* é declarada da seguinte forma: *HoughCircles(src gray, circles, method, dp, minDist, param1, param2, minRadius, maxRadius)* em que:

- **src gray:** imagem de entrada, neste caso, terá de estar em tons de cinza
- **circles:** vector de saída dos círculos encontrados
- **method:** método de deteção a usar
- **dp:** razão inversa do acumulador de resolução para a resolução da imagem
- **minDist:** distância mínima entre os centros e os círculos detetados
- **param1:** primeiro parâmetro de especificação do

método

- **param2:** segundo parâmetro de especificação do método
- **minRadius:** raio mínimo dos círculos
- **maxRadius:** raio máximo dos círculos

A função HoughLines é declarada da seguinte forma:  
`HoughLines(src canny, lines, rho, theta, threshold, srn, stn)` em que:

- **src canny:** imagem de entrada, convertida pelo algoritmo de Canny
- **lines:** vetor de saída das linhas encontradas
- **rho:** distância de resolução do acumulador nos pixéis.
- **theta:** Ângulo de resolut of the accumulator in radians
- **threshold:** parâmetro do acumulador de threshold
- **srn:** Para a Hough transform multi-scale, este é um divisor para a distância de resolução rho
- **stn:** Para a Hough transform multi-scale, este é um divisor para a distância de resolução theta

Depois de obtermos as linhas e os círculos das respectivas imagens, procedeu-se ao desenho dos mesmos na imagem. Em ambos os casos usou-se uma estrutura de dados do tipo *Ponto* e obteve-se as linhas, usando a função *line* em que aplicou os as linhas sobre a imagem resultante do Canny, e para os círculos usou-se a função *circle* que aplicou os círculos sobre a imagem original. Abaixo encontram-se os resultados respectivos a cada operação:

Com estes resultados, concluímos que usando os



Fig. 7 - Imagem com a deteção de círculos

um maior ou menor número de círculos na imagem.



Fig. 6 - Imagem com a deteção de linhas

algoritmos de Hough torna-se mais fácil detetar círculos e linhas. Também concluímos que no que toca à deteção de círculos, esta tornou-se mais fácil usando discos, muito possivelmente por estes serem círculos perfeitos o que ainda levou algum tempo para que conseguissemos visualizar o seu resultado. Alterando os parâmetros do algoritmo de deteção de círculos, podemos facilmente detetar