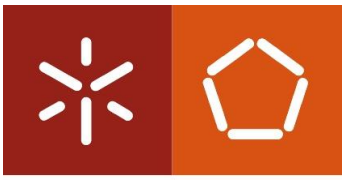

Universidade do Minho

Departamento de Informática

VC - Turorial 1

José Ferreira (A83683), Luís Pereira (A86265)

13 de dezembro de 2020

Conteúdo

1 Introdução **3**

2 Noise **4**

3 Smooth Images **5**
3.1 Spatial Smoothing para Ruído Salt-n-pepper 5
3.2 Spatial Smoothing para Ruído Gaussiano 6
3.3 Frequency Smoothing para Ruído Salt-n-pepper 7
3.4 Frequency Smoothing para Ruído Gaussiano 8

4 Detect Edges **9**
4.1 Canny Detector 9
4.2 Gaussian Smoothing 10
4.3 Gradient 11
4.4 NonMax 12
4.5 Double Threshold 13
4.6 Hysteresis Thresholding 14
4.7 Resultado Final e Comparações 14

Capítulo 1

Introdução

No âmbito da unidade curricular de Visão por Computador, foi-nos proposto implementar dois tipos de programas. Um deste programas tem como objetivo suavizar imagens. O outro outro tem a finalidade de detetar as bordas de uma imagem. O objetivo deste projeto consiste em colocar à prova os nossos conhecimentos da linguagem `MATLAB` e a aplicação da teoria leccionada nas aulas da UC.

Ao longo deste relatório iremos descrever os algoritmos implementados para cada um dos programas assim como relatar as diversas conclusões que obtivemos com os *outputs* dos programas criados.

Primeiro descrevemos como forma implementados dois geradores de ruído, salt-n-pepper e gaussiano. Estes algoritmos foram criados com o objetivo de testar os vários algoritmos que foram desenvolvidos.

Em seguida apresentamos os algoritmos de smoothing implementados apresentando várias comparações entre algoritmos de suavização implementados para os vários tipos de ruído criados.

Finalmente descrevemos como foi implementado um detetor de bordas. Para tal foi necessário a implementação de vários algoritmos para chegar ao resultado final, sendo estes Gaussian Smoothing, Gradient Calculation, Non-Maximum Supression, Double Threshold e Hysteresis Threshold. No final do capítulo é ainda comparado com outros métodos usando uma função do `MATLAB` de nome `edge`.

Capítulo 2

Noise

Para introduzir ruído numa imagem foi criado uma função denominada de ruído. Esta recebe uma matriz que representa uma imagem, o tipo de ruído a ser utilizado e os parâmetros para esse ruído.

O ruído gaussiano consiste em calcular uma matriz de valores aleatórios e adicionar essa matriz a matriz que representa a imagem. No caso deste ruído o parâmetro passado representa a variância dos valores calculados. Caso o ruído seja *salt-n-peppper* pixeis aleatórios na imagem são convertidos para branco ou para preto, neste caso o parâmetro representa a percentagem de ocorrência dos pixeis.

Figura 2.1: Original

Figura 2.2: Salt-n-pepper

Figura 2.3: Gaussian

Capítulo 3

Smooth Images

3.1 Spatial Smoothing para Ruído Salt-n-pepper

Ao longo desta secção iremos comparar os diversos tipos de smoothing espaciais implementados aplicando-os sobre o ruído salt-n-pepper descrito anteriormente.

Comparação entre diferentes tamanhos de kernel para smoothing espacial usando média. Como se pode observar, quanto maior o tamanho do kernel mais desfocada fica a imagem.

Figura 3.1: size: 5

Figura 3.2: size: 10

Figura 3.3: size: 20

Comparação entre diferentes tamanhos de kernel para smoothing com gauss. Neste caso, a diferença entre diversos tamanho de kernel torna-se inexistente. Tal deve-se ao facto de à medida que o tamanho do kernel aumenta os pixeis mais afastados tem um peso cada vez menor e por isso menor influencia sobre o pixel a ser tratado.

Figura 3.4: size: 5

Figura 3.5: size: 10

Figura 3.6: size: 20

Comparação entre diferentes tamanhos de kernel para smoothing com mediana. Neste caso,

quanto maior o tamanho do kernel mais detalhe se perde sendo que quando nos aproximamos do valor 20 a forma da imagem original fica irreconhecível. No entanto, No tamanho 5 do kernel todo o ruído salt-n-pepper desaparece.

Figura 3.7: size: 5

Figura 3.8: size: 10

Figura 3.9: size: 20

Assim, com base nas imagens acima apresentadas, o melhor tipo de smoothing para o ruído salt-n-pepper é o que utiliza a mediana.

3.2 Spatial Smoothing para Ruído Gaussiano

Ao longo desta secção iremos comparar os diversos tipos de smoothing espaciais implementados aplicando-os sobre o ruído gaussiano descrito anteriormente.

Comparação entre diferentes tamanhos de kernel para smoothing espacial usando media. Neste caso quanto maior o tamanho do kernel mais desfocado fica a imagem original.

Figura 3.10: size: 5

Figura 3.11: size: 10

Figura 3.12: size: 20

Comparação entre diferentes tamanhos de kernel para smoothing espacial usando gaussian. Neste caso a alteração do tamanho do kernel não modifica muito o resultado final.

Figura 3.13: size: 5

Figura 3.14: size: 10

Figura 3.15: size: 20

Comparação entre diferentes tamanhos de kernel para smoothing espacial usando mediana. Neste caso a imagem fica muito desfocada e quanto maior o tamanho do kernel mais detalhes se perdem.

Figura 3.16: size: 5

Figura 3.17: size: 10

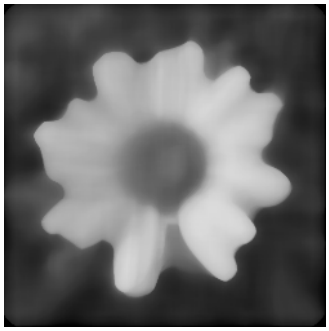

Figura 3.18: size: 20

Observando os resultados acima, o tipo de smoothing que reduz mais ruído sem perder detalhe acaba por ser o smoothing gaussiano.

3.3 Frequency Smoothing para Ruído Salt-n-pepper

Ao longo desta secção iremos comparar os diversos tipos de smoothing em frequência implementados aplicando-os sobre o ruído salt-n-pepper descrito anteriormente.

Comparação entre diferentes sigmas para smoothing por frequência usando gauss. Neste caso quanto maior o sigma melhor o resultado final da imagem.

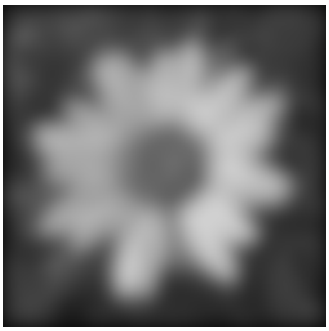

Figura 3.19: sigma: 10

Figura 3.20: sigma: 20

Figura 3.21: sigma: 50

Comparação entre diferentes cutoffs (com ordem fixa de 2) para smoothing por frequência usando Butterworth. Neste caso quanto maior o cutoff melhor a qualidade da imagem.

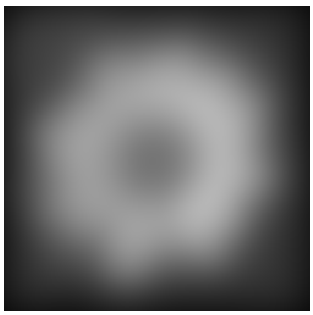

Figura 3.22: cutoff: 5

Figura 3.23: cutoff: 10

Figura 3.24: cutoff: 20

Neste caso o melhor filtro por frequência para o ruído gaussiano acaba por ser usando Butterworth.

3.4 Frequency Smoothing para Ruído Gaussiano

Ao longo desta secção iremos comparar os diversos tipos de smoothing em frequência implementados aplicando-os sobre o ruído gaussiano descrito anteriormente.

Comparação entre diferentes sigmas para smoothing por frequência usando gauss.

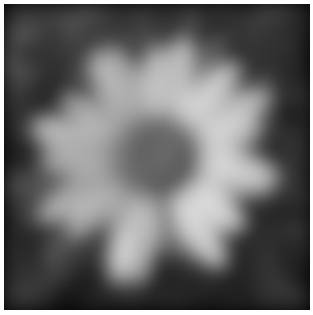

Figura 3.25: sigma: 10

Figura 3.26: sigma: 20

Figura 3.27: sigma: 50

Comparação entre diferentes cutoffs (com ordem fixa de 2) para smoothing por frequência usando Butterworth.

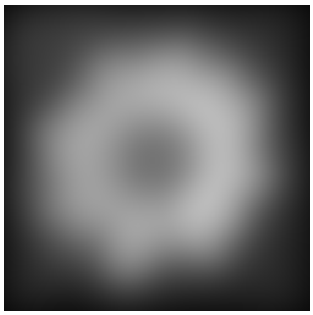

Figura 3.28: cutoff: 5

Figura 3.29: cutoff: 10

Figura 3.30: cutoff: 20

Neste caso o melhor filtro por frequência para o ruído gaussiano acaba por ser usando Gauss.

Capítulo 4

Detect Edges

4.1 Canny Detector

De modo a serem detetadas as bordas numa imagem foi implementado um detetor de bordas, tendo por base um desenvolvido por John F. Canny em 1986. Este utiliza um algoritmo multi-estágios com o objetivo de detetar várias bordas numa imagem.

John Canny também propôs como um detector de bordas deveria ser composto para ser considerado ideal:

- **Boa Detecção** - O algoritmo tem ser capaz de identificar todas as bordas possíveis numa imagem.
- **Boa Localização** - As bordas encontradas têm de se encontrar o mais próximo possível das bordas da imagem original.
- **Resposta Mínima** - Cada borda da imagem tem de ser marcada uma única vez. O ruído da imagem não pode criar falsas bordas.

Para a demonstração deste detetor foram escolhidos alguns *inputs*. Um dos *inputs* é uma imagem com ruído gaussiano. Naturalmente, para este valor foi escolhido um dos *outputs* do capítulo anterior, já que neste foi necessário adicionar ruído gaussiano a uma dada imagem.

Figura 4.1: Imagem com ruído gaussiano

Foram também escolhidos um **sigma**, um **x** e um **y**, usados para suavizar a imagem com ruído gaussiano. Serão explicados como estes interagem com a imagem mais à frente.

Antes do início da execução do detetor é acrescentado uma moldura preta à imagem, de tamanho x no eixo do x e de y no eixo do y, para que os kernels possam ser executados em

todos os pixeis. No entanto, é de notar que ao longo do algoritmo são criadas bordas falsas, que poderiam ser removidas com uma moldura superior, como o caso de uma moldura formada pelo estender da imagem com os pixeis que estão nas arestas da imagem.

4.2 Gaussian Smoothing

Como mencionado anteriormente, esta função necessita de todos os inputs dados a este programa. Com a pequena nuance da imagem já ir processada com a moldura. O objetivo desta função é suavizar a imagem tentando reduzir ao máximo o ruído, sem comprometer muito a qualidade da imagem. Com a imagem processada e os respectivos valores de sigma, x e y esta função cria o kernel gaussiano. O valor de sigma altera a potência deste filtro, quanto maior o seu valor mais a imagem fica suavizada, mas também fica com mais *blur*. Como se demonstra nas imagens abaixo com valores de sigma iguais a 0.8 e 1.8.

Figura 4.2: Imagem suavizada com sigma=0.8

Figura 4.3: Imagem suavizada com sigma=1.8

Depois existem os valores de x e y, estes simbolizam o tamanho do kernel, ou seja, quantos pixeis vão contribuir para a suavização do pixel a ser processado, desta forma se x e y forem iguais a 3 (size=3), o número dos pixeis adjacentes ao pixel que vão contribuir para o seu processamento são (3*3)-1=8. Claramente, os pixeis mais longe do pixel a ser processado terão menor contribuição para o seu valor final, como é possível identificar na imagem seguinte.

1/16

1	2	1
2	4	2
1	2	1

1/273

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

1/1003

0	0	1	2	1	0	0
0	3	13	22	13	3	0
1	13	59	97	59	13	1
2	22	97	159	97	22	2
1	13	59	97	59	13	1
0	3	13	22	13	3	0
0	0	1	2	1	0	0

Figura 4.4: Filtros Gaussianos de diferentes tamanhos

Quando aplicados estes kernels à imagem é possível obter as seguintes imagens para x e y iguais a 3 (size=3), e x e y iguais a 13 (size=13).

Figura 4.5: Imagem suavizada com size=3

Figura 4.6: Imagem suavizada com size=13

Consegue-se identificar que a moldura da imagem obtém tamanhos diferentes consoante o valor de x e y, aumentando quando estes também aumentam. Também se observa uma maior suavização quanto maior o kernel usado.

4.3 Gradient

De seguida a imagem processada na função anterior é enviada para a função Gradient. Como o nome indica, esta função é responsável pela computação dos gradientes verticais e horizontais da imagem, assim como as suas direções.

Este algoritmo tira proveito dos kernels de Sobel para realizar convoluções na imagem. É usado um kernel para a convolução no eixo do x, assim como outra no eixo do y.

X – Direction Kernel

-1	0	1
-2	0	2
-1	0	1

Y – Direction Kernel

-1	-2	-1
0	0	0
1	2	1

Figura 4.7: Filtros Gaussianos de diferentes tamanhos

Mais uma vez cada elemento da matriz vai especificar a importância dos pixeis adjacentes, para a convolução no pixel a ser processado. São então calculadas as convoluções nos eixos do x e y, resultando nas duas seguintes imagens.

Figura 4.8: Gradiente da imagem no eixo do x

Figura 4.9: Gradiente da imagem no eixo do y

Através destas é calculado a imagem gradiente através da hipotenusa entre as duas e dividindo o valor resultante pelo valor máximo dentre todos os pixeis que compõem a imagem resultante. É ainda calculada a matriz com as direções através do arco tangente entre as mesmas duas convoluções, para ser enviada à função seguinte.

Figura 4.10: Gradiente da imagem

4.4 NonMax

Esta função vai receber os dois *outputs* calculados pela função anterior, estes sendo a imagem gradiente e as respetivas direções. Inicialmente, vai pegar na matriz de direções e passa-las para os ângulos que estas representam, colocando-as numa nova matriz com esses valores. De seguida vai passar por todos os pixeis da imagem gradiente e encontra quais os pixeis que têm o valor máximo de intensidade na direção das bordas, utilizando a matriz calculada antes. Este processo vai transformar as bordas iniciais, que podiam ter bordas densas, em bordas mais finas, tendencialmente com apenas um pixel de espessura.

Como resultado desta função na imagem da flor, temos a seguinte imagem, onde realmente se consegue observar uma elevada diminuição na espessura das bordas especialmente nas pétalas da flor.

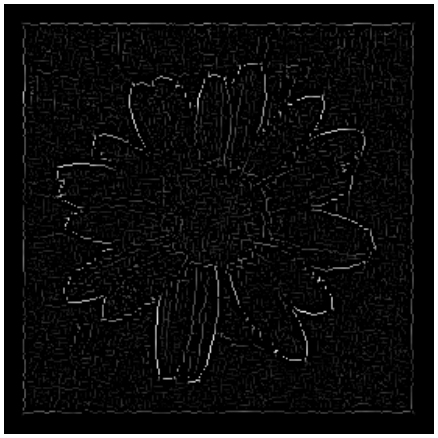

Figura 4.11: Imagem após NonMax Supression

4.5 Double Threshold

O objetivo desta função é identificar 3 tipos de pixeis, podendo estes ser **fortes**, **fracos** ou **não relevantes**.

- Pixeis **fortes** são pixeis que têm uma intensidade tão alta que se tem a certeza que vão fazer parte da borda final.
- Pixeis **fracos** são pixeis que têm uma intensidade que não é suficiente para ser considerados fortes, mas também não são tão fracos de modo a serem considerados não relevantes.
- Os pixeis **não relevantes** têm uma intensidade tão fraca que são retirados da imagem.

Para ser executado esta filtragem de pixeis é necessária a imagem calculada na função anterior, assim como um rácio para cálculo do *threshold* superior e outro para o *threshold* inferior. Através destes são calculados os verdadeiros *thresholds*. São então percorridos todos os pixeis da imagem comparando-os com os valores dos *thresholds* calculados, criando numa nova imagem com valores predefinidos, um valor para simbolizar pixel forte e outro para pixel fraco, e os não relevantes colocados a preto.

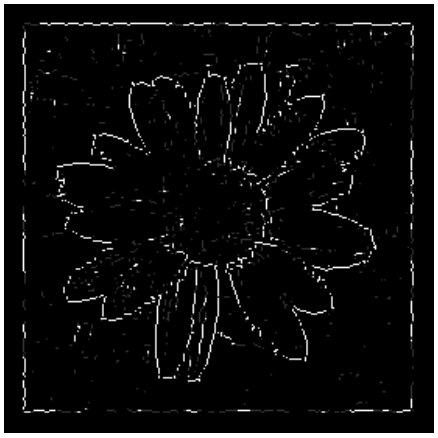

Figura 4.12: Imagem após double threshold

Os *thresholds* "ótimos"podem variar de imagem para imagem, de modo que se deve experimentar com vários valores para se conseguir ter os resultados mais próximos do pretendido. Ao baixar muito os *thresholds* começam a aparecer bordas a mais, ou então, ao subir os *thresholds* começam a desaparecer bordas importantes para a imagem.

4.6 Hysteresis Thresholding

A função Hysteresis Thresholding tem por base o resultado dos thresholds calculados na função anterior. Consiste em transformar alguns dos pixeis fracos em pixeis fortes e outros em pixeis pretos. Ao percorrer a imagem, sempre que encontra um pixel fraco são percorridos os pixeis em volta deste e se algum destes for forte, este também se torna num pixel forte, caso contrário este fica preto.

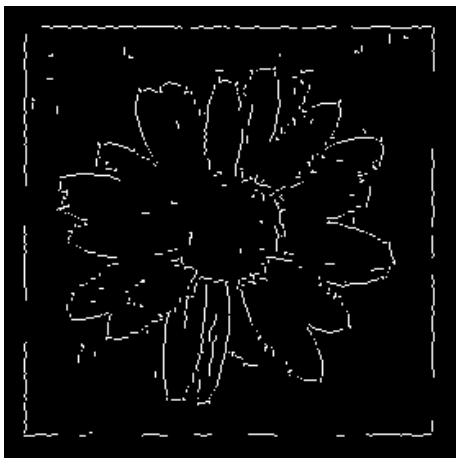

Figura 4.13: Imagem após hysteresis threshold

4.7 Resultado Final e Comparações

Após ser retirada à imagem a moldura que foi adicionada no início deste processo obtém-se o seguinte resultado. De notar que algumas bordas foram criadas devido à nossa implementação de moldura, uma implementação melhor suavizaria este problema.

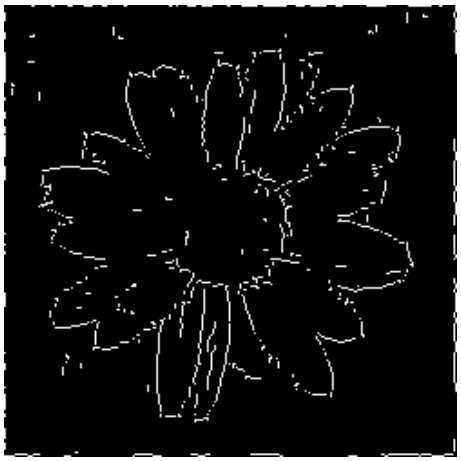

Figura 4.14: Imagem final sem moldura

Comparando com os outros métodos fornecidos pela função do `MATLAB` chamada de `edge()`, chega-se à conclusão que a nossa implementação do algoritmo é capaz de superar todas as outras, se os valores de *threshhold* forem bem definidos. Apesar de se termos testado a função `edge()` para o método Canny com vários valores, tanto de sigma como de *threshold*, apenas foi conseguido obter uma imagem completamente preta.

Figura 4.15: Sobel

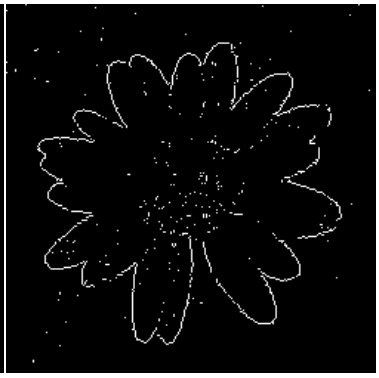

Figura 4.16: Prewitt

Figura 4.17: Roberts

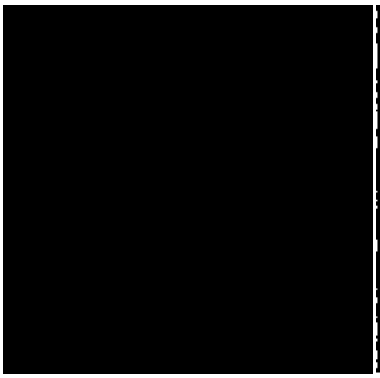

Figura 4.18: Canny

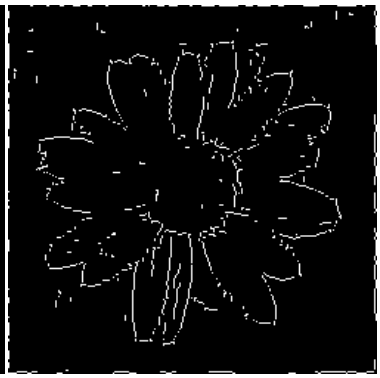

Figura 4.19: Nossa