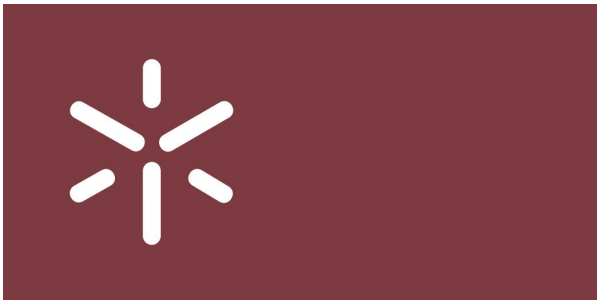


Universidade do Minho

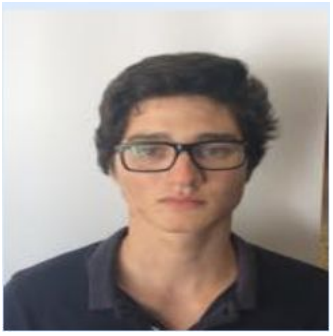
Mestrado Integrado em Engenharia Informática

Visão por Computador

Tutorial 1: Image Filtering and Edge Detection

Bernardo Viseu A74618

Luís Pereira A77667



João Neves A81366

10 Dezembro 2020

Índice

Introdução

Neste relatório vamos explicar a funcionalidade e a realização dos exercícios propostos no Tutorial 1 da unidade curricular Visão por Computador.

O objectivo deste projecto foi a realização de scripts e funções de *Matlab* que permitam fazer a filtragem e aplicar vários tipos de ruído a imagens.

Para alem disso, a 2 parte deste tutorial procura explicar de uma forma sucinta um algoritmo de detecção de bordas, canny edge detector, considerado um algoritmo multi-stage que detecta arestas que podem ser definidas como um conjunto de pixels conectados que formam um limite entre duas regiões de desarranjo.

Exercícios

Este tutorial encontra-se dividido em duas partes, neste tópico explicamos a realização dos vários tópicos pedidos para cada parte.

Parte 1: Design a program to smooth images

Neste primeiro exercício foi feito um script (*smoothfilters*) que chama uma função (*main_smoothfilters.m*). No script é feito o pedido dos parâmetros que o utilizador pretende aplicar a uma certa imagem, e na função é feito o algoritmo que modifica a imagem.

1. Inputs

Run Command para a Command Window do matlab:

```
filename='filename';smoothfilters
```

Atribui à variável filename a string com o nome da imagem a processar e inicia a script smoothfilters (a imagem tem de estar na pasta do script).
Exemplo:

```
filename='lena.jpg';smoothfilters
```

Se a imagem não estiver em grayscale, o script automaticamente aplica a função `image=rgb2gray(image);` de maneira a transformá-la em grayscale.

Ao executar a script serão pedidos vários inputs ao utilizador dentro da Command Window, as respostas podem ser dadas em letras maiúsculas, minúsculas ou mistura das duas. Algumas das perguntas contém um parâmetro default discriminado na pergunta, entre parênteses, caso o utilizador o queira usar deve somente carregar em enter.

Parâmetros existentes:

- Tipo de ruído ("*salt & pepper*" ou "*Gaussian*");
- Parâmetros de ruído;
    - Para *salt & pepper*: densidade de ruído,
    - Para *Gaussian*: média de ruído, variância.
- Tipo de domínio ("*spatial*" ou "*frequency*");
- Parâmetros de filtro;
    - Para *spatial:*
        - Tipo de filtro (*average, Gaussian* ou *median*);
        - Largura do filtro;
        - Para filtro *gaussian*: valor *standard deviation*;
    - Para *frequency:*
        - Tipo de filtro (*Gaussian* ou *Butterworth);*
        - Para filtro *Gaussian:* valor *standard deviation;*
        - Para filtro *Butterworth: filter order* e *cut-off frequency;*

Depois de estes parâmetros terem sido preenchidos, é então chamada a função *main_smoothfilters.m*, que vai receber todos os parâmetros de maneira a gerar as imagens com ruído e filtradas.

O script no entanto não acaba aqui: depois de ter as imagens geradas pela função, guarda-as com um nome adequado que indique os parâmetros e tipos de ruído/filtragem que foram aplicados, seguindo o formato "*OriginalName_smooth_filtertype_filterparameter.png".*

2. Filtros no domínio espacial

Aqui mostramos vários exemplos de imagens em que foi aplicado o algoritmo de ruído e filtragem, realizado na função *main_smoothfilters.m*.

Tendo em conta que existem dois tipos de ruído (*Salt & pepper* ou *Gaussian*), caso o domínio escolhido for o domínio **espacial**, pode se escolher entre os filtros *average*, *Gaussian* ou *median*:

Average filter em salt-and-pepper noise:

a.Salt-and-Pepper, 0.5 density

b. Average filter, Kernel size 5

c. Average filter, Kernel size 10

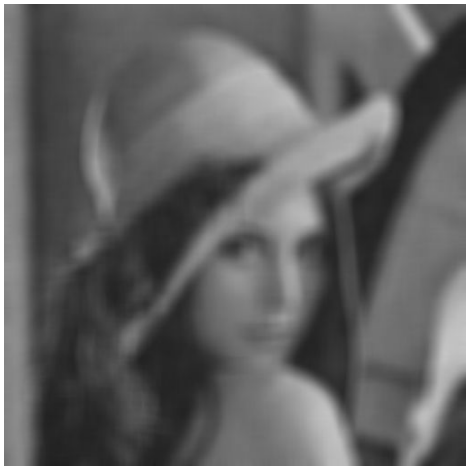
d. Average filter, Kernel size 20

Average filter em Gaussian noise:

a. Gaussian Noise, 0.01 variance

b. Average filter, Kernel size 5

c. Average filter, Kernel size 10



d. Average filter, Kernel size 20

Gaussian filter em salt-and-pepper noise:

a.Salt and Pepper, 0.5 density

b.Gaussian filter, Size 5, deviation 0.5

c.Gaussian filter, Size:20, deviation:0.5

d.Gaussian filter, Size 5, deviation 2

Gaussian filter em gaussian noise:

a. Gaussian Noise, 0.01 variance

b.Gaussian filter, Size 5, deviation 0.5

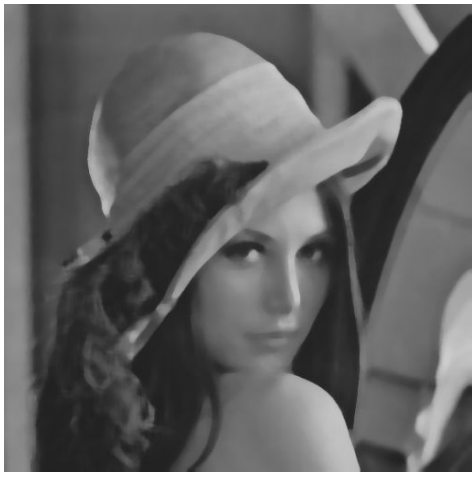
c.Gaussian filter, Size 20, deviation 1

d.Gaussian filter, Size 5, deviation 2

Median filter em salt-and-pepper noise:

a.Salt and Pepper, 0.5 density

b.Median filter, Size: 5



c.Median filter, Size: 10

d.Median filter, Size: 20

Median filter em Gaussian noise:

a.Gaussian Noise, 0.01 variance

b.Median filter, Size: 5

c.Median filter, Size: 10

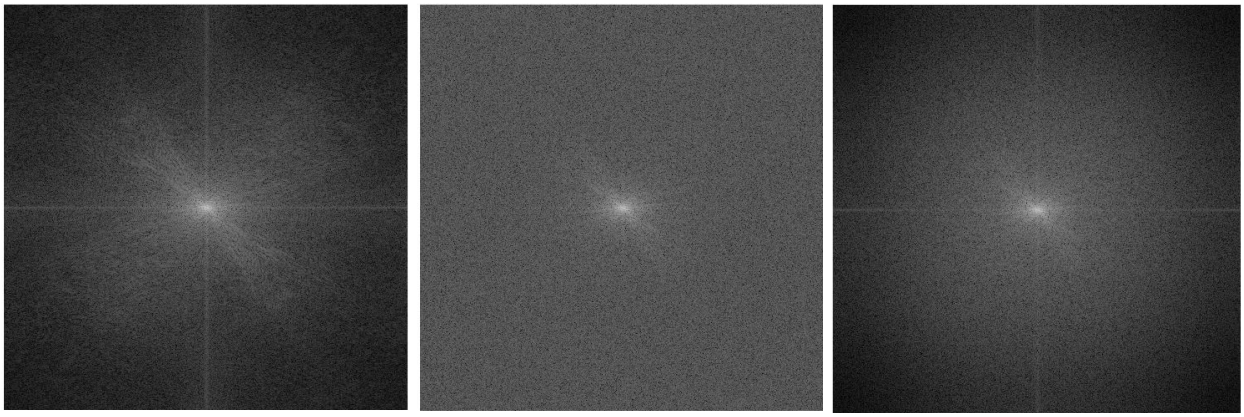
d.Median filter, Size: 20

A partir das imagens apresentadas, podemos concluir que:

Para imagens em que foi aplicado ruído *Gaussiano*, os melhores filtros foram os *average* e *median*. É de constatar que, com um *kernel* de tamanho inferior, não existe muito desfoque na imagem com qualquer filtro utilizado.

Para imagens em que foi aplicado ruído *Salt & Pepper,* o melhor filtro foi o *median*. Assim como em ruído *Gaussiano*, a aplicação de filtros com *kernel* inferior não provoca um grande desfoque na imagem, seja qual for o filtro utilizado.

3. Discrete Fast Fourier Transform (DFT)



a. Original b.Salt-and-Pepper Noise c.Gaussian filter,Size:3,dev:0.7

Nestas imagens podemos observar que o, na imagem em que foi aplicado o tipo de ruído *Salt & Pepper*, existe uma grande quantidade de ruído branco no espectro. O mesmo não acontece na aplicação do filtro Gaussiano.

4. Filtros no domínio da frequência

Na aplicação de um filtro no domínio de frequência, constatamos que, para um ruído *Salt & Pepper*, o filtro *Butterworth* apresentou o melhor resultado, equivalente ao filtro *median* em filtragem no domínio espacial.

a.Salt and Pepper, 0.5 density

b. Butterworth filter,Order:2, Cut:10

c. Butterworth filter,Order:0.5, Cut:10

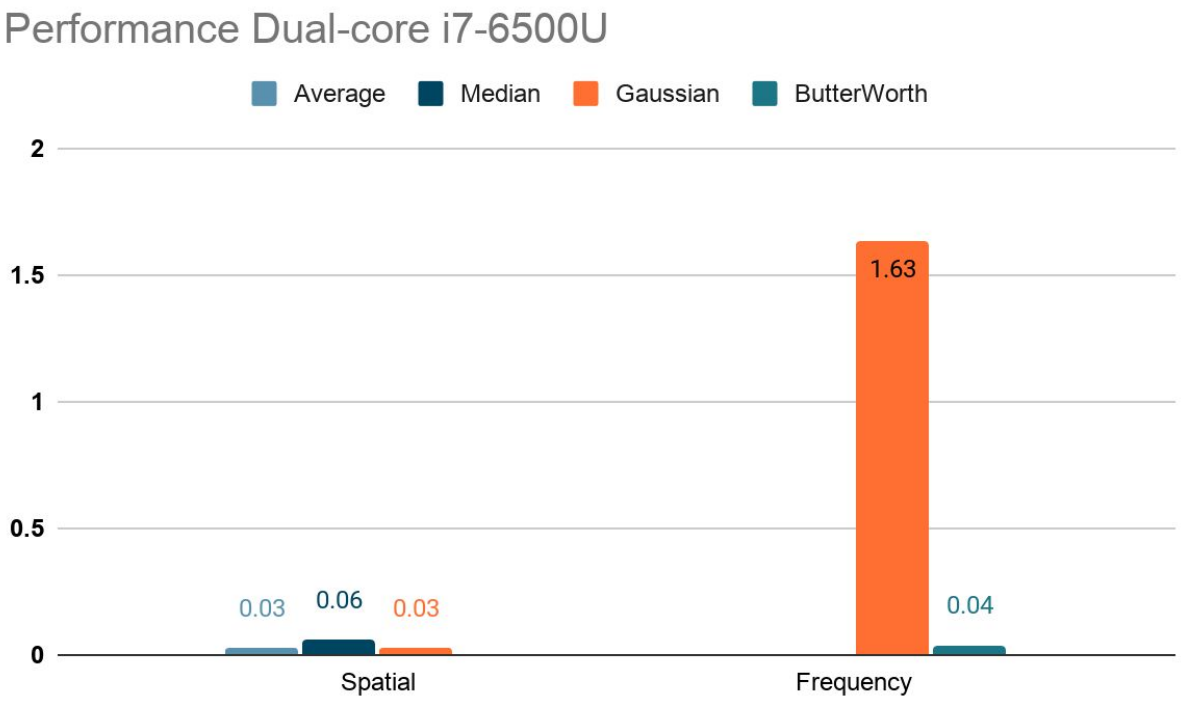
d. Butterworth filter,Order:0.5, Cut:50

Ringing effect

Caso seja aplicado um filtro com ordem muito elevada, é visível um *Ringing Effect,* nesta imagem em que utiliza um filtro *Butterworth* com ordem 50, já se destaca muito o efeito:

a.Butterworth Order 50

Performance



Para obter um estudo da *Performance* dos vários filtros, utilizamos a função de cputime do matlab, obtendo o tempo de cpu que demora a correr main_smoothfilters utilizando parâmetros de noise salt-and-pepper iguais. Os filtros mais eficientes foram average e gaussian no meio espacial, mas os mais precisos para este tipo de noise foram median e ButterWorth. Concluímos que para noise Sal-and-Pepper, devemos utilizar ButterWorth ou Median já que o impacto na performance são apenas milésimas de segundos e obtém resultados muito mais precisos ao real.

Parte 2: Canny Detector

5. Inputs

Nesta segunda parte, foi feito um script (*cannydetector*) que chama uma função (*main_cannydetector.m*). Assim como no primeiro exercício, no script é feito o pedido dos parâmetros que o utilizador pretende aplicar a uma certa imagem, e na função é feito o algoritmo que modifica a imagem.

Foi predefinido por nós, que a variância para a aplicação do filtro gaussiano não será definida pelo utilizador e será 0.12.

Posteriormente será pedido o tamanho da suavização gaussiana:

```
>> CannyDetector
Select the filterSize
5
```

Será também pedido a variância da suavização gaussiana:

```
Select the Variance
5
```

Por fim, será pedido para definir os limites, inferior e superior, do thresholding:

```
Select the LT
0.3*255
```

```
Select the UT
0.6*255
```

6. Gaussian Smoothing

De forma a escolher o melhor , foram feitos vários testes, chegando a conclusão que quanto mais se aumenta o σ, mais a imagem fica desfocada e menos ruído esta apresenta, visto que o gaussian smoothing, passa um filtro passa-baixo, que modifica cada pixel,.

7. Gradient

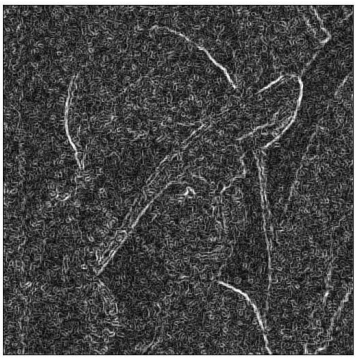
De forma a resolver a questão do gradiente, o operador de sobel foi fundamental, executando uma medição de gradiente espacial 2D numa imagem, enfatizando regiões de alta frequência espacial que correspondem a arestas.

Para além do referido acima, o algoritmo desenvolvido é também usado para localizar a magnitude do gradiente absoluto, assim como o ângulo dos gradientes direcionais. Segundo o operador de Sobel, as imagens podem ser traduzidas para as seguintes matrizes:

$$Z_h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad Z_v = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

A magnitude e o ângulo dos gradientes direcionais pode ser demonstrada atraves das seguintes equações:

$$\theta = \tan^{-1}\left(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x}\right) \quad \downarrow \qquad \|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \quad \downarrow$$

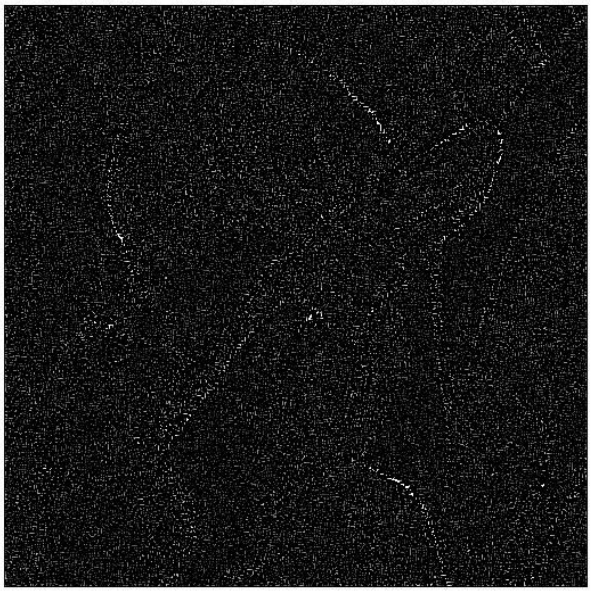


8. Non Maximum Suppression

O seguinte algoritmo tem como objetivo reduzir as bordas grossas produzidas pela magnitude da imagem e refinar a localização.

Para cada pixel da imagem do gradiente, o algoritmo:

1. Reconhece a direção da aresta
2. Especifica um número de orientações para o vetor gradiente
3. Compara a força da aresta do pixel onde está com os seus vizinhos, em dois lados opostos
4. Por fim, se a força da aresta do pixel onde se encontra for mais forte que os vizinhos, mediante a direção do gradiente, mantém o seu valor, caso contrário torna-se 0.



9. Double Thresholding

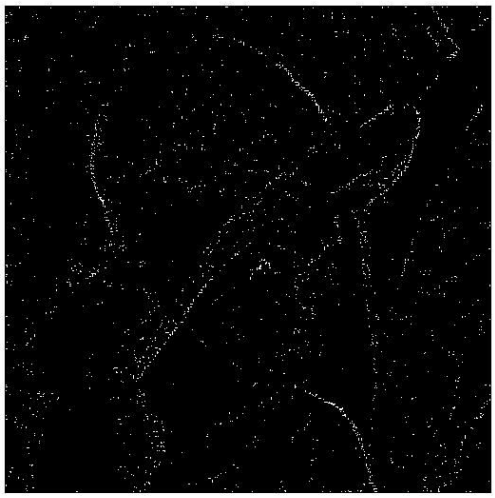
Após a análise do resultado do algoritmo Non Maximum Suppression, é possível perceber que algumas arestas podem não ser bordas e existe algum ruído na imagem.

De seguida, procedemos à utilização do operador Double Threshold, que aplica um filtro com o objetivo de remover,por exemplo, o ruído num específico alcance, por isso são definidos dois thresholds, o fraco e o forte.

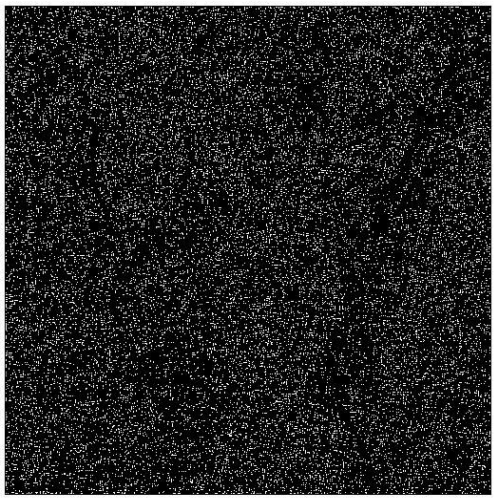
O algoritmo funciona da seguinte forma:

1. Pixels com valores abaixo do lower threshold são “thresholded out”que significa valor entre 0,-999..
2. Todos os valores acima do upper threshold são retidos.
3. Para todos os valores entre os thresholds:
    a. Valores que estão entre os dois thresholds e num raio específico de um ponto que não tenha sido modificado, permanecem inalterados.
    b. Todos os outros são identificados com um determinado threshold value.

Forte



Fraca

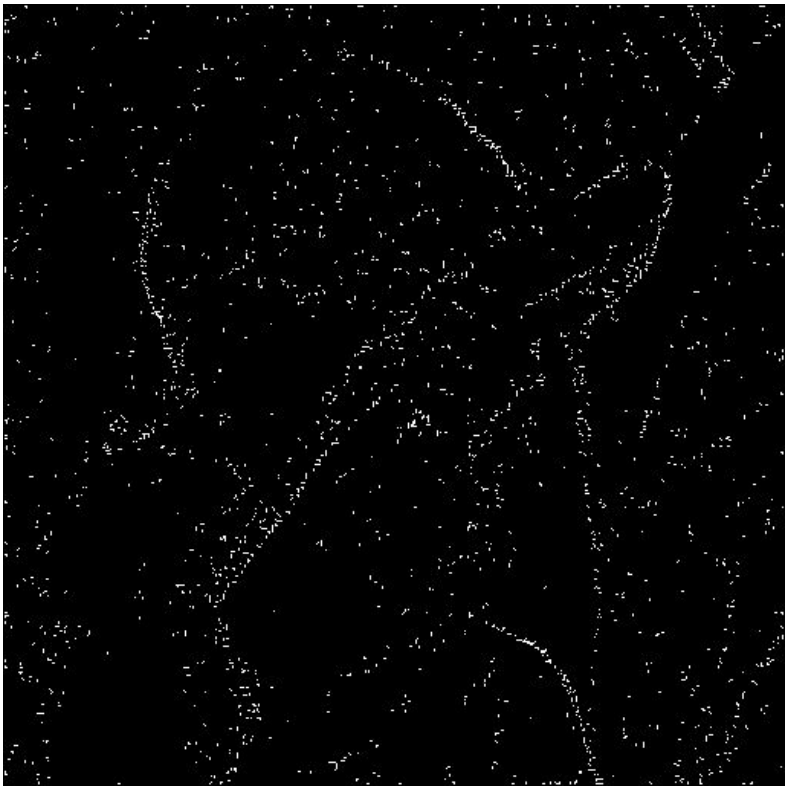


10. Edge Tracking by Hysteresis

Por fim, o Edge tracking by Hysteresis permite determinar quais das arestas fracas sao arestas reais.

As bordas fortes podem ser imediatamente incluídas na imagem final, e as bordas fracas serão incluídas somente se estiverem conectadas a uma aresta forte,para além disso, os limites definidos acima permitem que dificilmente o ruído e outras pequenas variações resultem numa borda forte.

Com isto, todas as arestas fracas, reais, estarão ligadas a uma aresta forte enquanto as respostas de ruído estarão, expectavelmente, desconectadas.



11. Comparison

Para finalizar, foi utilizada a biblioteca de edge detection do matlab, mais concretamente a função edge(), com o objetivo de comparar os vários operadores, Sobel, Prewitt, Laplacian of Gaussian e Canny, com a nossa implementação.Após analisar as imagens, abaixo demonstradas, podemos facilmente reparar que a não especificação de argumentos como o σ, , ou o kernel size, podem influenciar os resultados finais, por isso, em relação à nossa implementação, as imagens abaixo contem muitos mais pixels ligados, o que quer dizer que o nosso filtro, detecta e interpreta melhor quais das arestas são reais e quais devem ser eliminadas.

Sobel

Prewitt

Laplacian of Gaussian

Canny

Conclusão

Este trabalho mostrou ser muito satisfatório de realizar, devido à maneira como conseguimos ir testando os vários tipos de ruído e filtragem aplicados a imagens. Foi necessária pesquisa de vários conceitos da linguagem de programação Matlab, assim como um bom entendimento da matéria ensinadas nas aulas, os slides providenciados pela professora foram então essenciais para a execução deste projeto.

Sentimos-nos satisfeitos com o resultado final, termos conseguido atingir os objetivos propostos no Tutorial, e ganhamos um melhor entendimento do conteúdo desta cadeira.