

IST - Summer Internships 2020

## **Reconhecimento de Moedas**

Vasco Araújo

C2TN - Centro de Ciências e Tecnologias Nucleares

---

## 1 Descrição do Problema

As moedas têm, maioritariamente devido à utilização diária, riscos e defeitos (alguns não são facilmente observáveis a olho nu). Da mesma forma, moedas antigas também apresentam este tipo de imperfeições, algumas das quais podem estar relacionadas com a sua história, p.e. enterramento, limpeza, etc.

Estas imperfeições, mesmo que não desejadas pelos colecionadores ou proprietários, podem servir como identificação única de cada moeda.

Durante o estágio, várias moedas similares foram fotografadas usando o método RTI (Reflectance Transformation Imaging). Posteriormente, as fotografias foram processadas para obter o padrão de riscos/imperfeições característicos de cada moeda. O objectivo final é poder distinguir a partir do padrão calculado para cada moeda, as diferentes moedas fotografadas e poder identificar cada uma delas.

## 2 Soluções

Antes de começar a tentar solucionar o problema, tive que aprender o que era o método RTI e como funcionava. O site da [Cultural Heritage Imaging](#) é um bom sítio por começar.

Após isso, usando ficheiros RTI já feitos antes de eu chegar, fui experimentar os vários filtros no software utilizado para visualizar os ficheiros RTI, o RTIViewer.

Intuitivamente, olhando para os filtros presentes no RTIViewer, o filtro que melhor parecia preservar e acentuar os riscos e imperfeições de cada moeda seria o Specular Enhancement. No entanto, pareceu-me boa ideia experimentar todos para ter a certeza. Antes da minha chegada, tinham sido produzidos ficheiros .xmp para os seguintes filtros: Default, Diffuse Gain, Specular Enhancement e Normal Unsharp Masking, logo foram esses os escolhidos para um teste mais a fundo.

À partida, a minha primeira ideia foi usar técnicas e algoritmos de *Edge Detection* para detectar as imperfeições da moeda. Na [documentação](#) do MATLAB estão apresentados todos os algoritmos possíveis. Após ter testado todos os algoritmos de Edge Detection (Figura 1), cheguei à conclusão que

---

o Canny seria o mais eficaz. Lendo [este](#) artigo comprova-se que, não sendo eficiência computacional uma preocupação o Canny produziria os melhores resultados, e assim sendo foi o que usei daí para a frente.

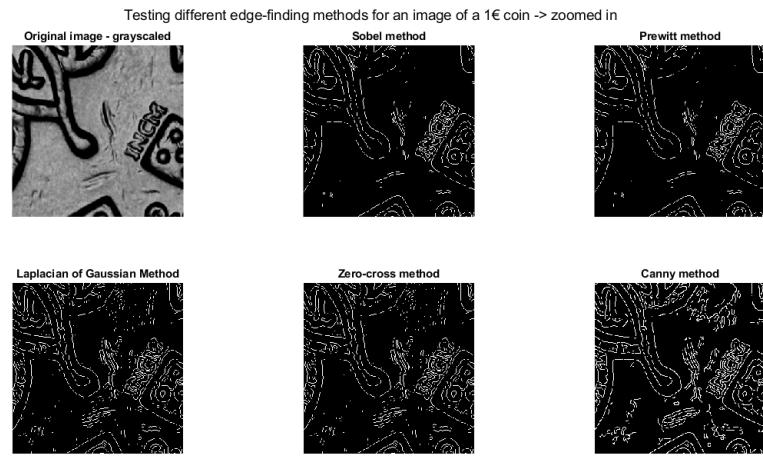


Figura 1: Aplicação de vários algoritmos de *Edge Detection* à mesma moeda

Nas Figuras [2](#) e [3](#) podemos observar o resultado da aplicação do algoritmo Canny na mesma moeda aplicando diferentes filtros no RTIViewer sem nenhum tratamento de imagem adicional.

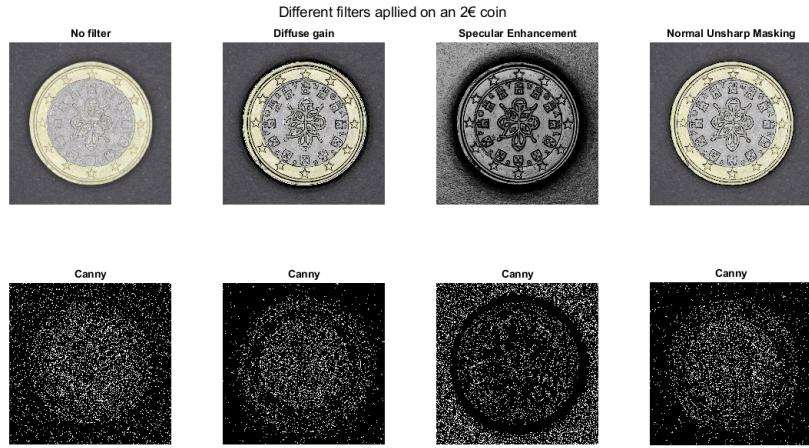


Figura 2: Aplicação do algoritmo Canny a imagens da mesma moeda com diferentes filtros do RTIViewer aplicados

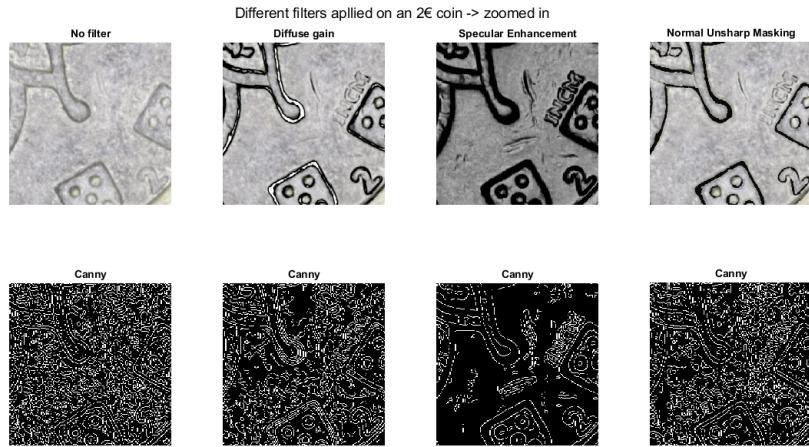


Figura 3: Aplicação do algoritmo Canny a imagens da mesma moeda com diferentes filtros do RTIViewer aplicados com zoom

Como podemos verificar, o filtro aplicado à moeda no RTIViewer que tem melhor resultados é o Specular Enhancement e é o que será usado a partir de

---

agora. No entanto, alguns problemas saltam à vista. Na Figura 2 podemos verificar que a imagem tem bastante ruído fora do círculo da moeda, e pela Figura 3 verifica-se que o Canny detecta bastantes mais *edges* do que necessário, por vezes detectando onde não existe nada.

Para resolver o problema do ruído, desenvolvi uma função `clearOutsideCoin()` para remover o ruído fora da moeda, calculando o raio da moeda e preenchendo todo o espaço fora do círculo de píxeis pretos. O código dessa função encontra-se no Anexo 3.1. Uma limitação da função `clearOutsideCoin()` é que para calcular o raio correcto da moeda, tem um *loop for* por todos os possíveis raios e escolhe o primeiro que se encontrar no limite  $]600;700[$ , que foram valores aos quais eu cheguei pela experiência, mas obviamente uma moeda fotografada com um zoom muito diferente do que usei terá um raio diferente e a função não irá calcular o raio correctamente. Infelizmente não encontrei uma maneira de calcular o raio correcto de uma forma robusta a todas as situações. Isto é certamente algo a melhorar.

Para resolver o problema do algoritmo Canny apresentar mais traços do que o necessário, irei simular o algoritmo Canny com algumas variações para tentar um melhor resultado. Os passos em pseudocódigo do algoritmo podem ser vistos [aqui](#).

Após a função `clearOutsideCoin()` irei começar por aplicar um [filtro gaussiano](#) para reduzir o ruído na imagem resultante. Considerando a documentação do MATLAB verifica-se que podemos controlar o valor de sigma da função. Quanto maior o sigma mais turva fica a imagem e portanto mais detalhe perde, o que também reduz os arranhões desnecessários. Para tentar escolher qual o valor de sigma a usar irei testar para valores de 0, 1, 2, 3 e 4.

O resultado do filtro gaussiano será enviado para uma função `ScratchDetection()` (pode ser visto no Anexo 3.2) que faz uma série de operações morfológicas na imagem de maneira a isolar as imperfeições da moeda. De maneira a detectar as *edges* da moeda irei calcular o gradiente de magnitude da imagem, e para isso uso a função do MATLAB [imgradient](#). A seguir, queremos segmentar a imagem para criar uma imagem binária apenas com *edges* e não *edges*, e para isso temos que escolher um determinado valor de *threshold*. Para calcular o valor de *threshold* a aplicar na imagem o [método de Otsu](#) parece ser o indicado pois calcula automaticamente o melhor valor para cada imagem. Após isso é uma questão de [morfologicamente fechar](#) a imagem e remover todos os objectos (com o objecto refiro-me a um conjunto de píxeis interligados)

---

inferiores a x número de píxeis e superiores a y (experimentalmente cheguei aos valores de 100 e 1500, respectivamente). Obviamente este método não é robusto por duas razões: i) como está dependente do número de píxeis definidos no código, qualquer imagem que fuja um pouco à norma irá resultar numa solução errada ii) o acto de morfologicamente unir os objectos e remover os maiores objectos tem a limitação de remover as imperfeições que se encontram perto de características da moeda, como por exemplo as estrelas ou os castelos da moeda. Apesar destas limitações, em algumas moedas as imperfeições obtidas eram bastante satisfatórias como pode ser observado nas Figuras 4 e 5.

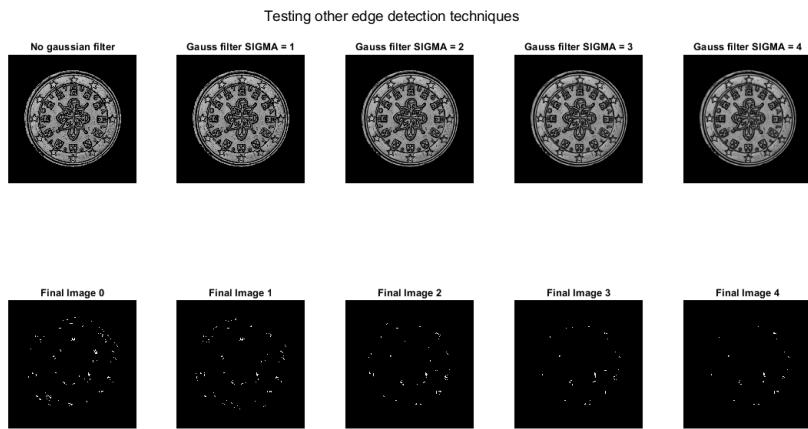


Figura 4: Recriação do algorimo Canny à mesma moeda com diferentes valores de sigma no filtro gaussiano

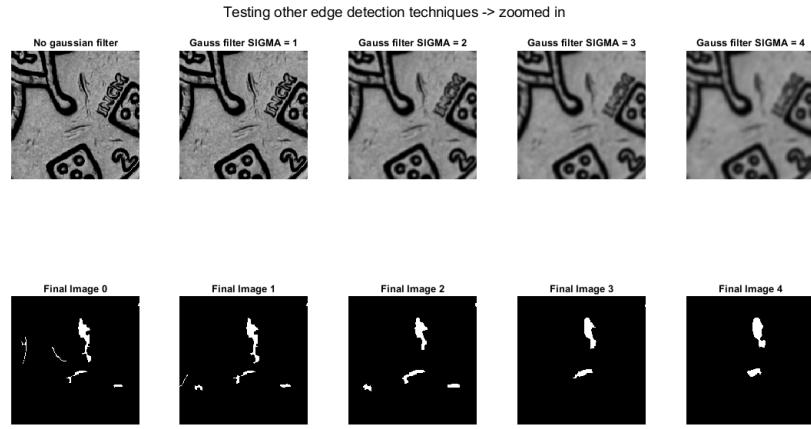


Figura 5: Recriação do algoritmo Canny à mesma moeda com diferentes valores de sigma no filtro gaussiano com zoom

Para melhor visualizar o resultado sobrepus as imperfeições calculadas na Figura 5 em cima da imagem inicial e o resultado pode observar-se na Figura 6.

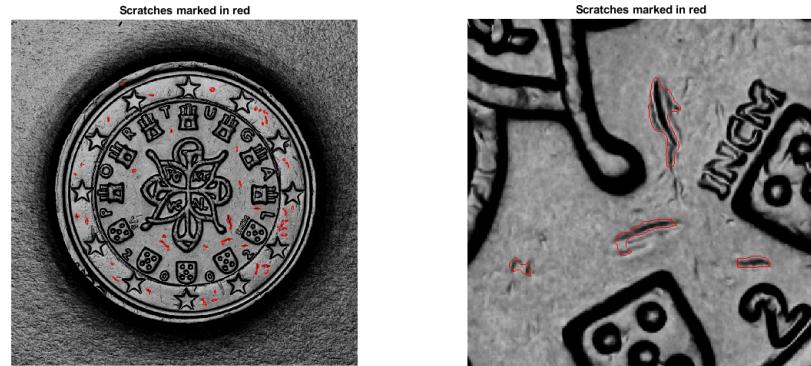


Figura 6: Sobreposição das falhas encontradas com a recriação do algoritmo Canny na moeda original

---

Para tentar resolver estes problemas, fui tentando uma série de modificações ao código, mudando alguns valores, ou outros métodos de *Edge detection*, tais como [bottom-hat filtering](#), [top-hat filtering](#) ou a [transformada de Hough](#), sem grandes resultados, pois nenhuma maneira era suficientemente robusta para todas as moedas que estava a testar na altura, e na altura só estava a testar com 6 moedas.

Fui então por outro caminho. Lembrei-me de tentar fazer uma subtracção, ou seja, computar uma imagem da moeda "ideal" - sem riscos nem imperfeições - e subtrair a moeda em questão a essa moeda ideal, ficando apenas com tudo o que não estivesse presente na moeda ideal como resultado final.

Para tal, tinha que computar a imagem da moeda "ideal". A função `createBinaryEdge2()` (pode ser vista no Anexo [3.3](#)) recebe a imagem RGB da moeda original e devolve uma reconstrução binária dos traços mais carregados da moeda. Com um número razoável de moedas pode-se então fazer a média das imagens binárias resultantes dessa função.

No entanto, ao fazer a média das 6 moedas iniciais apercebi-me que as moedas não estavam todas exactamente orientadas no mesmo ângulo, nem exactamente centradas.

Para resolver o primeiro problema era necessário orientar as moedas todas na vertical. Para isso, pensei orientar todas as imagens pela ponta da estrela superior. No entanto, analisando as moedas, as estrelas nem sempre eram perfeitas, ou devido a erros na cunhagem da moeda ou devido à construção do ficheiro `.ptm`, para além de que seria necessário escolher orientar apenas pela estrela superior, o que apesar de poder ser possível iria ser complicado. Após pesquisa na internet decidi usar um método não muito convencional: o utilizador teria que desenhar uma linha no eixo pelo qual queria orientar a moeda, e usando a transformada de Hough para calcular qual a orientação dessa linha, a moeda seria rodada nesse eixo. Este método tem a clara desvantagem de ser necessário o utilizador ter que desenhar a linha na moeda, para além de introduzir erros humanos pois uma pessoa não consegue desenhar várias linhas na orientação exacta. Apesar disso, os resultados foram bastante bons e não consegui arranjar um método melhor. No futuro esta é outra coisa a melhorar. O código da função pode ser analisado no Anexo [3.4](#).

Estando já as moedas orientadas na mesma direcção era necessário encontrar uma maneira de centrar as moedas, de forma a poder calcular a média de

---

uma forma minimamente precisa. Para isso achei que a melhor maneira seria simplesmente recortar a imagem de maneira a ficar apenas com a moeda. O código pode ser analisado no Anexo 3.5. Usando a função `imfindcircles()` do MATLAB, procurando para ”*Object Polarity : bright*”, cortamos a imagem com base no círculo preto exterior logo a seguir às estrelas, e assim o cálculo da média será muito mais preciso.

Para fazer a média só nos falta fazer com que cada imagem tenha o mesmo tamanho de pixels, pois caso contrário a soma das imagens daria erro. Usando a função `imresize` para pôr todas as moedas com o número de pixels da imagem mais pequena, depois é uma questão de somar as imagens todas e dividir pelo número de imagens.

Usando as primeiras 6 moedas, os resultados da rotação, recorte da moeda e da média pode ser observado nas Figuras 7 e 8.

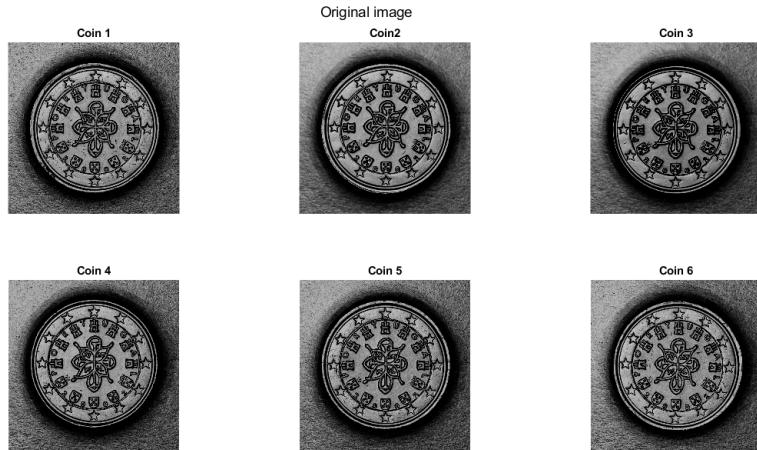


Figura 7: Imagens de 6 moedas obtidas com o filtro Specular Enhancement no RTIViewer

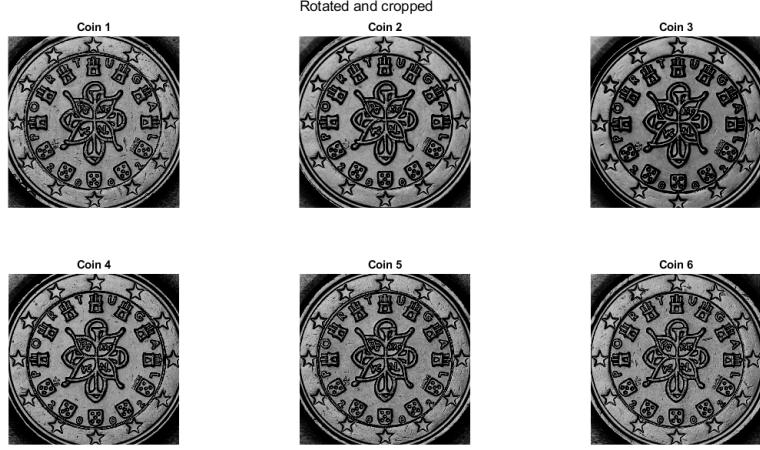


Figura 8: As mesmas moedas da Figura 7 orientadas na vertical e recortadas

No cálculo da média, de maneira a ficarmos com uma imagem mais nítida usamos um *threshold* de 0.33 após a divisão pelo número total de imagens. De forma a minimizar pequenos erros usamos a função *thicken* e por fim fechamos morfologicamente. O resultado da média pode ser observado na Figura 9.

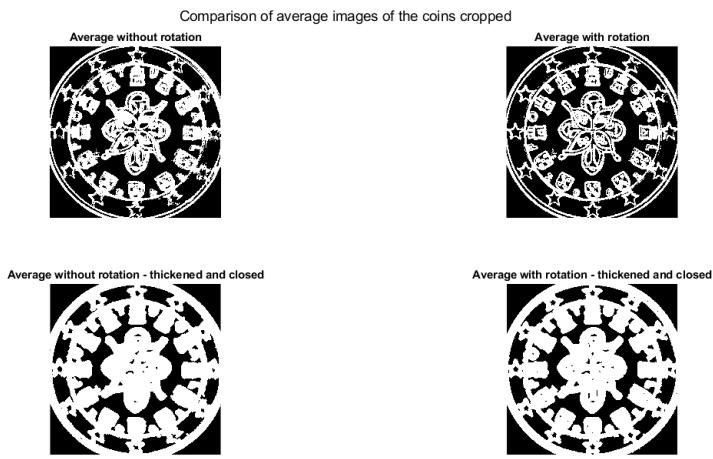


Figura 9: Média das 6 moedas da Figura 7, não recortadas e recortadas

---

A seguir, era necessário fazer uma imagem binária da moeda que queríamos subtrair. Primeiro tentei utilizar técnicas mais convencionais de *Edge Detection*, analisando as várias funções presentes na biblioteca do MATLAB. No entanto, estes algoritmos nunca funcionavam muito bem pois nem sempre era fácil para os algoritmos detectar as características menos pronunciadas da moeda, como por exemplo a coroa INCM ou as letras VS. Mas depois apercebi-me que bastaria segmentar a imagem com base na cor, pois no fundo o que se pretendia encontrar eram apenas com os traços mais intensos da moeda, ou seja, numa imagem a preto e branco, apenas ficar com os pretos mais carregados da moeda. Para fiz uma função que, com uma imagem em *grayscale* apenas devolvesse com 1 os traços da moeda inferiores a x valor, e 0 para os restantes. Essa função, `createBinaryImage()`, pode ser analisada no Anexo [3.6](#).

Voltando a usar a função `imresize` para pôr as moedas com o mesmo tamanho, já podíamos efectuar a subtracção das imagens binárias. Experimentalmente verifiquei que, apenas usando a média o resultado nem sempre era consistente, pois apesar de as moedas serem recortadas e orientadas, havia ainda pequenos erros, até porque a própria cunhagem das moedas nem sempre é ideal, e portanto para moedas um pouco mais diferentes da média o resultado final era um pouco estranho.

Por isso, decidi fazer outra subtracção para além da feita com a média. Essa subtracção seria feita entre a imagem binária resultante da função `createBinaryImage()` e uma imagem binária com apenas os traços principais da moeda sem as imperfeições. Tentei usar a função `createBinaryEdge2()` para computar essa imagem, mas o resultado dessa função continha algumas imperfeições, o que não tem grande problema para o cálculo da média pois essas imperfeições são desprezáveis para uma quantidade razoável, mas para apenas uma moeda o resultado iria levar a erros no final. Logo fiz outra função, `createBinaryEdge()`, que usando a função `bwboundaries` cria outra imagem binária apenas com os traços principais da moeda.

Os resultados das três funções, aplicado à primeira moeda pode ser observado na Figura [10](#).

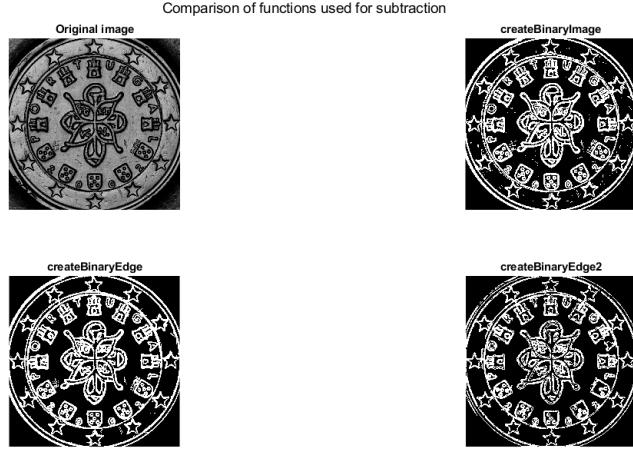


Figura 10: Comparação das três funções usadas para a subtração binária

Após isto, já podemos efectuar as subtrações. As operações binárias serão as seguintes:

$$\begin{aligned} \text{SUB1} &= \text{AND}(\text{binaryImage}, \text{NOT}(\text{binaryEdge})); \\ \text{SUB2} &= \text{AND}(\text{binaryImage}, \text{NOT}(\text{average})); \\ \text{SUB3} &= \text{AND}(\text{SUB1}, \text{SUB2}); \end{aligned}$$

Onde `binaryImage` é o retorno da `createBinaryImage()`, `binaryEdge` o retorno da `createBinaryEdge()` e `average` a média calculada usando a `createBinaryEdge2()`.

Na Figura 11 podemos observar um *flowchart* bastante simplificado do funcionamento do programa e da utilização das diversas funções e subtrações.

Apresenta-se na Figura 12 o resultado de todas as operações binárias na primeira moeda.

O problema que este programa tinha era que não era nada *user-friendly*, pois era necessário escrever no MATLAB o nome dos ficheiros, e se se quisesse fazer para outras imagens era necessário ir ao *script* alterar o que lá estava. Logo, o que era preciso era um programa em que o utilizador pudesse escolher aquilo que queria fazer, apresentando uma série de opções, e permitir que ele escolhesse as imagens que queria analisar.

Usando vários botões de selecção e a função `uigetfile` para seleccionar as

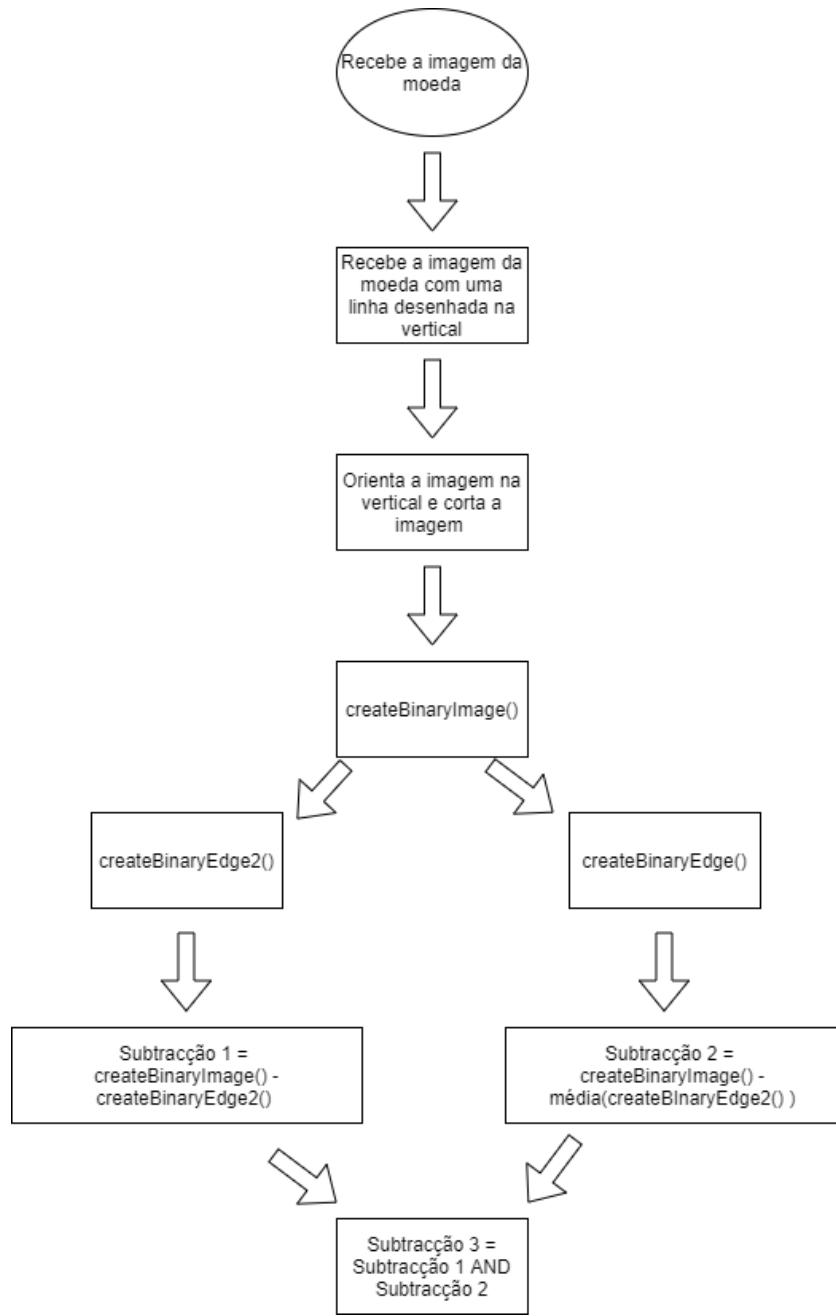


Figura 11: *Flowchart* simplificado do funcionamento das três subtrações binárias

---

Coin 1



Figura 12: Operações binárias à primeira moeda

imagens escolhidas pelo utilizador, depois era uma simples questão de alterar o programa para correr o algoritmo em loop por todas as imagens escolhidas pelo utilizador.

As opções que podem ser seleccionadas são: i) ver x número de maiores imperfeições ii) ver todos os arranhões iii) verificar se a moeda existe no dataset iv) estimar qual o valor da moeda.

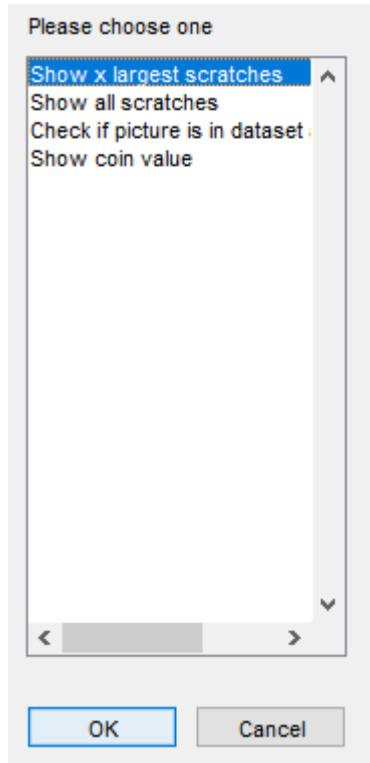


Figura 13: Menu que mostra as opções possíveis de serem escolhidas pelo utilizador após processamento da imagem

Devido à alteração do programa para poder correr qualquer número de moedas, era necessário alterar a forma como era calculada a média das imagens. Anteriormente o programa corria apenas para 6 moedas e a média era calculada usando essas 6 moedas. No entanto, era benéfico que à medida que o utilizador fosse escolhendo novas moedas que a média se alterasse devido à nova informação disponível. Para além disso, se um dos objectivos era distinguir as várias moedas e criar um padrão para cada, era necessário criar uma "base de dados" para guardar a informação de cada moeda.

Para criar a base de dados achei que a maneira mais simples seria criar um ficheiro .mat para guardar a informação para cada moeda. Criei dois ficheiros, um chamado "binaryCoinsForAverage.mat", que guardava a matriz binária que era utilizada para calcular a média das moedas, e um chamado "binarySub3.mat", que guardava a matriz binária sub3, pois achei que esta

---

seria a melhor maneira para comparar duas moedas, já que continha apenas a informação das suas imperfeições, o que é um padrão único para cada moeda.

A sua implementação é muito simples: primeiro, o programa verifica se os ficheiros existem. Se existirem, guarda as suas informações em variáveis para poderem ser usados no programa, e se não cria um novo ficheiro. No final do programa, se o utilizador seleccionar a opção para verificar se a moeda existe na base de dados, o programa vai comparar a matriz binária da moeda seleccionada pelo utilizador às matrizes presentes na base de dados. Se alguma coincidir, significa que a moeda já se encontra na base de dados e é apresentada ao utilizador uma mensagem a avisar, e se não existir adiciona a moeda ao fim da base de dados. O código dessa função, `checkEqualPercentage()` pode ser observado no Anexo 3.7. No entanto, ao correr o programa verifiquei que a matriz binária resultante da SUB3 não era muito eficaz, pois como a matriz era praticamente toda preta, já que só as imperfeições estavam a branco, a semelhança entre qualquer moeda era enorme, superior a 98% para quase todas. Logo, decidi usar a informação da matriz usada para calcular a média para verificar a semelhança entre as moedas, o que garantiu um resultado bastante melhor.

Neste momento o programa já realizava duas tarefas. Podia mostrar todas as imperfeições da moeda a vermelho na imagem inserida pelo utilizador, e verificar se a moeda já existia na base de dados, e se não, acrescentar a moeda à base de dados existente. Após isso, como no artigo as principais imperfeições da moeda eram assinaladas através de um círculo vermelho, lembrei-me de implementar isso no programa. Era uma questão de, com base na matriz SUB3, escolher os x maiores objectos, calcular o seu centro usando a função `regionprops` e desenhar à parte um círculo nesse ponto. O código dessa função, `drawRedCircle2()`, pode ser visto no Anexo 3.8.

A seguir, lembrei-me de tentar calcular o valor facial da moeda, distinguido entre 1 e 2 euros. Como as fotografias estudadas são da parte de trás da moeda, tentar ler o valor facial da moeda não era possível, logo era preciso ir pelas cores. A moeda de 2 euros tem o círculo interior dourado e o círculo exterior prateado, enquanto que na moeda de 1 euro é ao contrário. Por tentativa e erro, encontrei um intervalo para os pontos RGB correspondentes à cor dourada. Com a percentagem de pontos dourados na moeda é simples de descobrir o valor da moeda. O código dessa função pode ser observado no Anexo 3.9.

---

Por fim, para poder usar mais as funcionalidades do método RTI, quis fazer com que o utilizador pudesse inserir várias fotografias da mesma moeda mas iluminada de diferentes ângulos como sistema RTI para extrair mais informação. Se, a título de exemplo, o utilizador introduzisse uma fotografia iluminada no ponto  $(0,0)$  e mais quatro fotografias, uma em cada quadrante, o programa iria correr o algoritmo para cada uma das fotografias e no final somar as imperfeições encontradas em cada fotografia, para uma análise mais completa dos riscos da moeda. Somando as imperfeições de cada moeda obtém-se uma visão muito mais completa dos riscos da moeda, e é aqui que se pode ver com toda a clareza as possibilidades da técnica de RTI, como se pode observar nas Figuras [14](#) e [15](#).



Figura 14: Moeda 1 só com imagem iluminada na coordenada  $(0,0)$  utilizando o RTIViewer



Figura 15: Moeda 1 com imagem na coordenada (0,0) e uma fotografia iluminada por cada quadrante utilizando o RTIViewer

Por último, como tínhamos um número considerável de ficheiros RTI de moedas espanholas e francesas achei que seria boa ideia fazer uma forma de criar ficheiros para a média das moedas para cada país, pois se se juntasse uma moeda portuguesa com uma francesa, por exemplo, a média sairia bastante errada para qualquer uma das moedas. Para além de Portugal, Espanha e França adicionei a hipótese de o utilizador escolher moedas de outros países, que não entrariam para nenhuma média.

De seguida apresentam-se as janelas que são mostradas ao utilizador ao correr o programa.

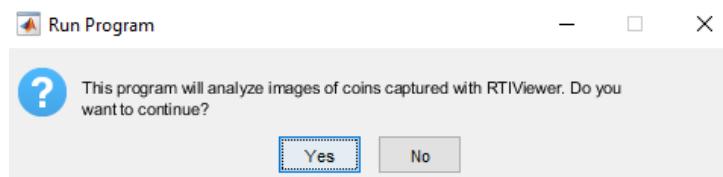


Figura 16: Janelas apresentadas ao utilizador ao correr o programa : 1



Figura 17: Janelas apresentadas ao utilizador ao correr o programa : 2

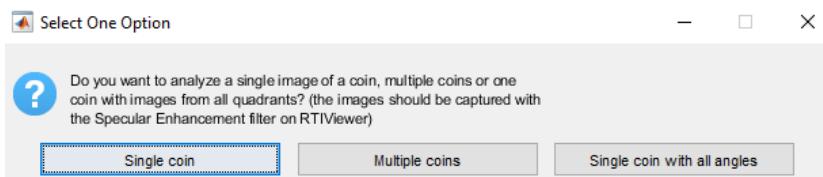


Figura 18: Janelas apresentadas ao utilizador ao correr o programa : 3

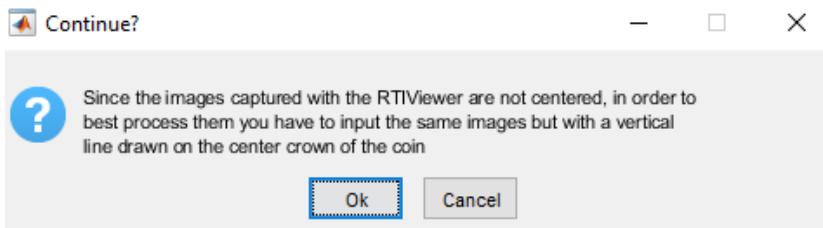


Figura 19: Janelas apresentadas ao utilizador ao correr o programa : 4

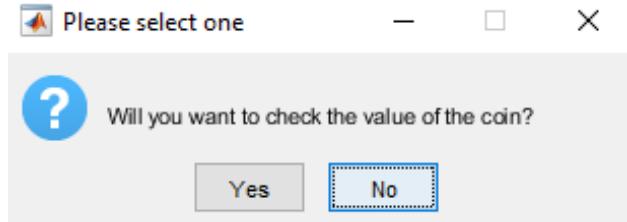


Figura 20: Janelas apresentadas ao utilizador ao correr o programa : 5



Figura 21: Janelas apresentadas ao utilizador ao correr o programa : 6

## 2.1 Conclusões

No fim, penso que o trabalho foi concluído com sucesso.

As principais falhas/coisas a melhorar são:

- i) o cálculo do raio da moeda, e consequentemente o recorte da moeda, pois há vezes em que o programa não é robusto o suficiente, sobretudo na questão de achar o semi-círculo preto no qual cortar, pois isso depende de valores

---

inseridos e que, consoante o grau de zoom com que a fotografia foi tirada, podem estar errados

- ii) o facto de que para orientar as moedas na vertical de modo à média ser fidedigna é necessário que o utilizador use um software à parte para desenhar a linha por cima da moeda, o que para além de ser um incómodo gera pequenos erros pois a linha desenhada pelo utilizador nunca será perfeita
- iii) com um grau elevado de moedas adicionadas à média, a média deixa de ser muito robusta. Por experiência própria reparei que cerca de 6 imagens é o ideal para o cálculo da média
- iv) por cada moeda ser diferente, nenhum parâmetro é ideal para todas as moedas. Valores como o número de *thickening* ou a sensibilidade para encontrar os círculos pretos devem ser mudados consoante a moeda em questão, os valores que pus são simplesmente aqueles que produzem melhores resultados para a maioria das moedas

---

### 3 Anexos

#### 3.1 Anexo 1 - clearOutsideCoin()

```
function [specularCleared, radiiCoin, centerCoinX, centerCoinY] =
    clearOutsideCoin(specular)
% This function receives a RGB image of a coin and returns an image
    % with the
% outside of the coin entirely black to reduce noise in the image

% Turns image into grayscale
specularG = rgb2gray(specular);

% Turns the grayscaled image into a binary image
binary = imbinarize(specularG);
% Closes the image to make the contrast bigger
bwclose = imclose(binary, strel('disk', 20));

% Calculate the
stats = regionprops('table', bwclose, 'Centroid', 'MajorAxisLength
    ', 'MinorAxisLength');
centers = stats.Centroid;
diameters = mean([stats.MajorAxisLength stats.MinorAxisLength],2);
radii = diameters/2;

coinI = 0;
for i = 1:size(radii,1)
    if radii(i) > 600 && radii(i) < 700
        coinI = i;
    end
end
if coinI == 0
    [~,coinI] = max(radii);
end
centerCoinX = centers(coinI,1);
centerCoinY = centers(coinI,2);
radiiCoin = radii(coinI);

circleImage = false(size(specularG,1), size(specularG,2));
```

---

```
[x, y] = meshgrid(1:size(specularG,1), 1:size(specularG,2));
circleImage((x - centerCoinX).^2 + (y - centerCoinY).^2 <=
    radiiCoin.^2) = true;

specularG(~circleImage) = 0;
specularCleared = specularG;

end
```

---

### 3.2 Anexo 2 - ScratchDetection()

```
function finalImage = ScratchDetection(specularG)

% Finds gradient magnitude and direction of an image
[Gmag, ~] = imgradient(specularG);

% Using Otsu's method to automatically calculate the best threshold
% value
% that maximizes the interclass variance of pixels
t = graythresh(specularG);

% Thresholds gradient magnitude by the value calculated before
Gmag = Gmag > t*max(Gmag(:));

% Morphologically closes image
closeGmag = imclose(Gmag, strel('disk',5));

% Removes objects smaller than 100pixels from image
remove = bwareaopen(closeGmag, 100);

% Again closes the images to join the lines together
newCloseGmag = imclose(remove, strel('disk',3));

% Removes objects larger than 1500 pixels
finalImage = bwpropfilt(newCloseGmag,'Area',[0 1500]);

end
```

---

### 3.3 Anexo 3 - createBinaryEdge2()

```
function binaryImage = createBinaryEdge2(im)
% This function receives an RGB image and returns a binary matrix
    % with only
% the biggest edges of the coin

% Paints the outside of the coin black
[imClear, radiiCoin, centerCoinX, centerCoinY] = clearOutsideCoin(
    im);

aux = imClear;

% Every pixel below 75 in grayscale will be converted to pure white
    % in
% order to make the contrast bigger to the rest of the coin
t = 75;
imClear(aux < t) = 255;

aux = imClear;

imClear(aux ~= 255) = 0;

circleImage = false(size(imClear,1), size(imClear,2));

[x, y] = meshgrid(1:size(imClear,1), 1:size(imClear,2));

circleImage((x - centerCoinX).^2 + (y - centerCoinY).^2 <=
    radiiCoin.^2) = true;

imClear(~circleImage) = 0;

binaryImage = imClear;

binaryImage = imclose(binaryImage, strel('disk', 2));

% Removes objects smaller than 200 pixels to reduce noise and small
    % errors
binaryImage = bwareaopen(binaryImage, 200);
```

---

end

---

### 3.4 Anexo 4 - rotatesImageHough()

```
function rotatedImage = rotatesImageHough(im, im_line)
% This function receives the original RGB image and returns
% the original image rotated along the vertical line

% Binarizes the image
binaryEdge = createBinaryEdge(im_line);

% Hough transform
[H, theta] = hough(binaryEdge);

% Finds peaks in hough transform
peak = houghpeaks(H);

% Calculates the angle of the line
barAngle = theta(peak(2));

% Rotates the image without a line accordingly to the orientation
% of the
% line
rotatedImage = imrotate(im,barAngle, 'bilinear','crop');

end
```

---

### 3.5 Anexo 5 - cropsCircle2()

```
function [croppedImage, cropX, cropY, diameter] = cropsCircle2(im)
%This function receives a RGB image and crops to stay only the
    circle of
%the coin

% Turns the image from RGB to grayscale
specularG = rgb2gray(im);

% Estimates the coin radii
[~, radiiCoin, centerCoinX, centerCoinY] = clearOutsideCoin(im);

% Rounds the radii towards a positive integer
radiiCoin = ceil(radiiCoin);

% Finds all circles
[centers, radii] = imfindcircles(specularG, [radiiCoin-120
    radiiCoin-60], 'Sensitivity', 0.99, 'ObjectPolarity', 'bright');

% We want the smallest radii found
[~,index] = min(radii);
if ~isempty(radii)
    centerCoinX = ceil(centers(index,1));
    centerCoinY = ceil(centers(index,2));
    radiiCoin = ceil(radii(index));

end

% Returns the cropped image
croppedImage = imcrop(im, [centerCoinX-radiiCoin centerCoinY-
    radiiCoin 2*radiiCoin 2*radiiCoin]);

% This are for when the flagAllAngles = 1, otherwise they wont be
% necessary
cropX = centerCoinX-radiiCoin;
cropY = centerCoinY-radiiCoin;
diameter = 2*radiiCoin;
end
```

---

### 3.6 Anexo 6 - createBinaryImage()

```
function binaryImage = createBinaryImage(im)
% This function receives a RGBimage and returns the image in black
    and
% white, with everything black except the countours inside the coin

% Clears outside of the coin and gets radii and center of coin
[imClear, radiiCoin, centerCoinX, centerCoinY] = clearOutsideCoin(
    im);

aux = imClear;

% Every pixel below 75 turns to white
t = 75;
imClear(aux < t) = 255;

aux = imClear;

% Every pixel that is not white turns to black
imClear(aux ~= 255) = 0;

circleImage = false(size(imClear,1), size(imClear,2));

[x, y] = meshgrid(1:size(imClear,1), 1:size(imClear,2));

circleImage((x - centerCoinX).^2 + (y - centerCoinY).^2 <=
    radiiCoin.^2) = true;

% Every pixel outside of the coin turns to white
imClear(~circleImage) = 0;

% Variable to return
binaryImage = imClear;

end
```

---

### 3.7 Anexo 7 - checkEqualPercentage()

```
function percentage = checkEqualPercentage(matrix1, matrix2)
% This function receives two matrices and returns the percentage of
    their similarity

% This program will only run if the matrices are the same size
if size(matrix1,1) == size(matrix2,1) && size(matrix1,2) == size(
    matrix2,2)
    equalNum = 0;
    totalNum = size(matrix1,1) * size(matrix1,2);
    % Loops the entire matrix
    for i = 1:size(matrix1,1)
        for j = 1:size(matrix1,2)
            if matrix1(i,j) == matrix2(i,j)
                equalNum = equalNum + 1;
            end
        end
    end
    % Calculates percentage to return
    percentage = (equalNum / totalNum) * 100;
end

end
```

---

### 3.8 Anexo 8 - drawRedCircle2()

```
function centers= drawRedCircle2(sub, numberCircles, numberImages)
% This function receives the final image with only the
    imperfections in white and everything else in black, finds the
    three largest imperfections and returns it's center to plot red
    circles around it

for i = 1:numberImages
    stats{i} = regionprops('table', sub{i}, 'Area', 'Centroid');

    [~, maxIndex{i}] = maxk(stats{i}.Area,numberCircles);

    centers{i} = stats{i}.Centroid(maxIndex{i},:);
end

end
```

---

### 3.9 Anexo 9 - checkCoinValue()

```
function isOneEuro = checkCoinValue(im)
% This function receives an RGB image of a coin and calculates the
    percentage of gold colour the coin has. If os less than a
    certain value, the coin is of 1 euro, if not is 2 euro

% Crops the image so that the size of the coin will be the same,
    regardless
% of the zoom used on the original image
croppedImage = cropsCircle2(im);

% Converts the image to HSV colormap
hsvImage = rgb2hsv(croppedImage);

% Calculates the gold points
goldPoints = hsvImage(:,:1) <= 0.2 & hsvImage(:,:2) >= 0.1 &
    hsvImage(:,:3) >= 0.8;

% Calculates the percentage of gold points on the coin
percentGold = 100*(sum(sum(goldPoints))/(size(croppedImage,1)*size(
    croppedImage,2)))

% Returns the coin value (1 if is 1 euro, 0 if is 2 euro)
if percentGold < 33
    isOneEuro = 1;
else
    isOneEuro = 0;
end

end
```