Visão Por Computador
Trabalho 2

Christophe Oliveira nº2011154912
Noé Godinho nº 2011159459

9 de Março de 2015

Conteúdo

Introdução

O objectivo deste trabalho é implementar os algoritmos de detecção de três tipos de pontos característicos numa imagem, sendo eles:

- Cantos;
- Rectas;
- Circunferências.

A primeira detecção, a detecção de cantos, é a mais simples de realizar, sendo necessário para cada ponto $\boldsymbol{P}$, uma vizinhança $\boldsymbol{Q}$ e uma matriz $\boldsymbol{C}$ definida por uns somatórios obtidos a partir da vizinhança e procedendo à sua diagonalização, como iremos ver mais à frente.

A detecção de rectas e circunferências, é mais complicado, o que implica a utilização da transformada de Hough que, no caso das rectas, envolve o uso da equação polar.

Exercício 1

O primeiro algoritmo a implementar é o algoritmo de detecção de cantos de uma imagem $\boldsymbol{I}$, considerando uma vizinhança $\boldsymbol{Q}$ de dimensão $2N+1\times 2N+1$. Também é necessário definir um valor σ para λ_2, acima do qual se considera a existência de um canto.

Este algoritmo fornece uma lista de pontos com $\lambda \geq \sigma$ e cujas vizinhanças não se sobrepõem.

1.1

De maneira a ser possível implementar o algoritmo, é necessário calcular as componentes **X** e **Y** do gradiente em toda a imagem $\boldsymbol{I}$ previamente convertida em escala de cinzento pela função *rgb2gray* do Matlab.

```
Sobel_hor_mask = [-1 -2 -1; 0 0 0; 1 2 1];
Sobel_ver_mask = [-1 0 1; -2 0 2; -1 0 1];

Ix = imfilter(image, Sobel_hor_mask);
Iy = imfilter(image, Sobel_ver_mask);

Ix = double(Ix);
Iy = double(Iy);
```

Com este código, obtém-se duas matrizes, cada uma com o tamanho da imagem, com o gradiente correspondente ao eixo **X** e **Y**, respectivamente *Ix* e *Iy*.

1.2

Uma vez calculado o gradiente, percorre-se cada ponto $\boldsymbol{P}$ da imagem $\boldsymbol{I}$ e realiza-se uma série de operações.

1.2.1

Para cada ponto $\boldsymbol{P}$ é necessário obter a matriz $\boldsymbol{C}$.

Esta matriz é obtida da seguinte forma:

$$\boldsymbol{C} = \begin{bmatrix} \sum I_x^2 & \sum I_x I_y \\ \sum I_x I_y & \sum I_y^2 \end{bmatrix}$$

Em que I_x e I_y são as matrizes obtidas na alínea anterior.

Para cada ponto $\boldsymbol{P}$, não são utilizados todos os pontos das matrizes, mas as submatrizes pertencentes à vizinhança $\boldsymbol{Q}$ de dimensão $2N + 1 \times 2N + 1$ do ponto $\boldsymbol{P}$ para os cálculos do somatório.

Como a matriz é simétrica, só é necessário calcular três somatórios, sendo um deles reutilizável em dois pontos da matriz.

O código correspondente ao cálculo de cada somatório é o seguinte:

```
Q_Ix_Iy = sum(sum(Ix(px - limit : px + limit, py - limit : py + limit) .*
                  Iy(px - limit : px + limit, py - limit : py + limit)));
Q_Ix_2 = sum(sum(Ix(px - limit : px + limit, py - limit : py + limit).^2));
Q_Iy_2 = sum(sum(Iy(px - limit : px + limit, py - limit : py + limit).^2));
```

Em que *px* é o ponto $\boldsymbol{P}$ actual e *limit* é o cálculo do limite da vizinhança $\boldsymbol{Q}$, definido por $2N + 1 \times 2N + 1$, sendo N fornecido pelo utilizador.

De salientar que é necessário realizar uma soma dupla, já que o Matlab calcula primeiro a soma de cada linha e a insere num array, só depois é calculada a soma do array final, contendo todos os pontos da submatriz calculada.

1.2.2

Depois de ter a matriz $\boldsymbol{C}$, é necessário calcular os valores próprios de $\boldsymbol{C}$ e selecionar o valor mais baixo de λ_2.

O cálculo dos valores próprios de $\boldsymbol{C}$ é feito pela diagonalização através da rotação dos eixos de coordenadas, já que $\boldsymbol{C}$ é uma matriz simétrica.
Depois de realizar a diagonalização, obtém-se a seguinte matriz:

$$\boldsymbol{C} = \begin{vmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{vmatrix}$$

Em que λ_1 e λ_2 correspondem aos valores próprios de $\boldsymbol{C}$.

Para calcular a diagonalização de $\boldsymbol{C}$, foi usado o seguinte código:

```
C = eig([Q_Ix_2 Q_Ix_Iy; Q_Ix_Iy Q_Iy_2]);
```

Uma vez calculada a diagonalização de $\boldsymbol{C}$, é necessário calcular o valor mínimo de λ_2.

Sabe-se que $\lambda_1 \geq \lambda_2$, logo um canto é definido pela localização do ponto $\boldsymbol{P}$ onde o valor próprio de λ_2 é suficientemente elevado, sendo determinado por um σ.

Como o valor que é necessário obter é o menor valor de λ_2, e o menor valor será sempre λ_2 pela regra acima explicada, o código para obter esse valor é o seguinte:

```
minimum = min(C);
```

1.2.3

Depois de ter sido calculado o menor valor de λ_2, é necessário verificar se este é suficientemente elevado para ser considerado um canto.

O valor de verificação é determinado por σ, sendo este, fornecido pelo utilizador.

O código que trata dessa verificação é o seguinte:

```
if minimum >= sigma
    list = [list; minimum px py];
end
```

Caso o valor seja superior a *sigma*, o valor é suficientemente elevado para ser considerado um canto, logo é armazenado na lista que contém todos os cantos que são cantos, tal como as suas coordenadas.

1.3

Uma vez obtida a lista $\boldsymbol{L}$ com todos os valores de λ_2 que satisfazem a condição, é necessário ordenar a lista de pontos por ordem decrescente dos valores de λ_2.

```
list = sortrows(list, -1);
```

A função *sortrows*, quando enviado um valor negativo, ordena por ordem decrescente, uma lista/array/matriz na coluna indicada pelo valor.

1.4

Por último, para a implementação correcta do algoritmo, percorre-se a lista $\boldsymbol{L}$ ponto a ponto e, para cada ponto $\boldsymbol{P}$ existente na mesma, removem-se os que pertencem à sua área de vizinhança.

Desta maneira, assegura-se que não são detectados vértices com pontos de vizinhança comuns.

```
for iterator = 1 : x
    %saves the point existent in the iteration
    px = list(iterator, 2);
    py = list(iterator, 3);

    %verifies if it exists some point in the list which belongs to the neighborhood
    firstPoint = exists(list(1:iterator, 2), px - limit:px + limit);
    secondPoint = exists(list(1:iterator, 3), py - limit:py + limit);

    %if some point belongs in the neighborhood, then it's saved in the
    %list of deleted points to be deleted later
    if firstPoint == 1 && secondPoint == 1
         deleteFromList = [deleteFromList; iterator];
    end
end

list(deleteFromList, :) = [];
```

Este ciclo percorre os todos os pontos da lista $\boldsymbol{L}$, verifica se existe algum ponto com coordenadas pertencentes à vizinhança $\boldsymbol{Q}$ na lista, com a função *exists*, uma função que, quando encontra um ponto em comum, retorna 1. Depois esse ponto é inserido numa lista temporária, que contém todos os pontos a eliminar, para serem removidos no final do ciclo.

Exercício 2

Como foi explicado anteriormente, para que valor mínimo de λ_2 num ponto $\boldsymbol{P}$ seja considerado canto, é necessário definir um valor adequado de σ para obter o melhor resultado possível.

Uma maneira de obter isso, como pedido neste exercício, é usar o histograma de todos os valores de λ_2 existentes na imagem. Dessa maneira, podemos verificar qual o valor mais adequado para σ.

```
[n, h] = hist(histogram_values, 15);
bar(h, n);
```

Exercício 3

Uma vez feito o algoritmo, é necessário executá-lo para dois tipos de imagens diferentes, obter o seu histograma, para poder escolher melhor um sigma, mostrar o tensor da matriz C para os cantos escolhidos e mostrar a imagem final com os cantos detectados.

Neste caso, será testada a imagem "chess2.png"com vizinhanças 3x3 e 5x5.

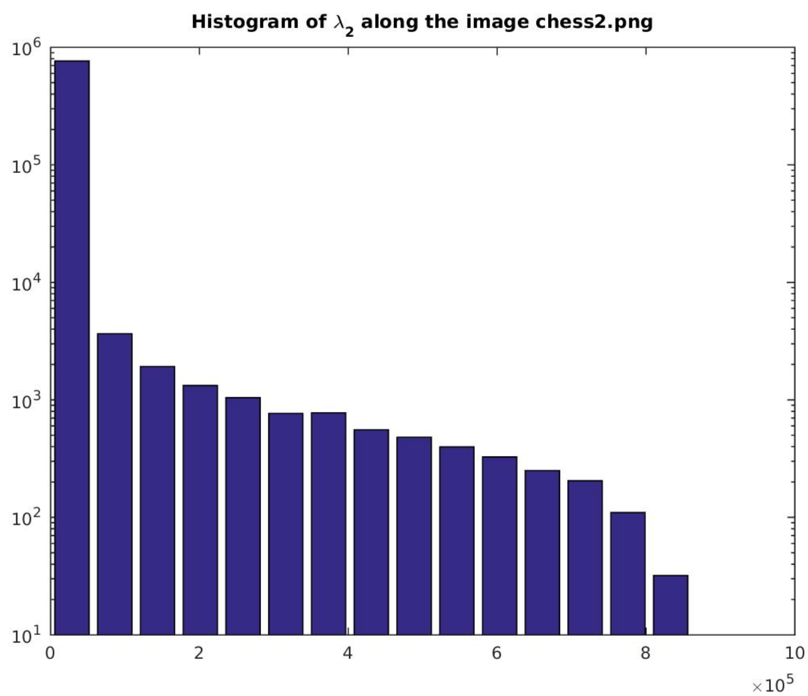

Figura 1: Histograma de λ_2 da imagem chess2, vizinhança 3x3

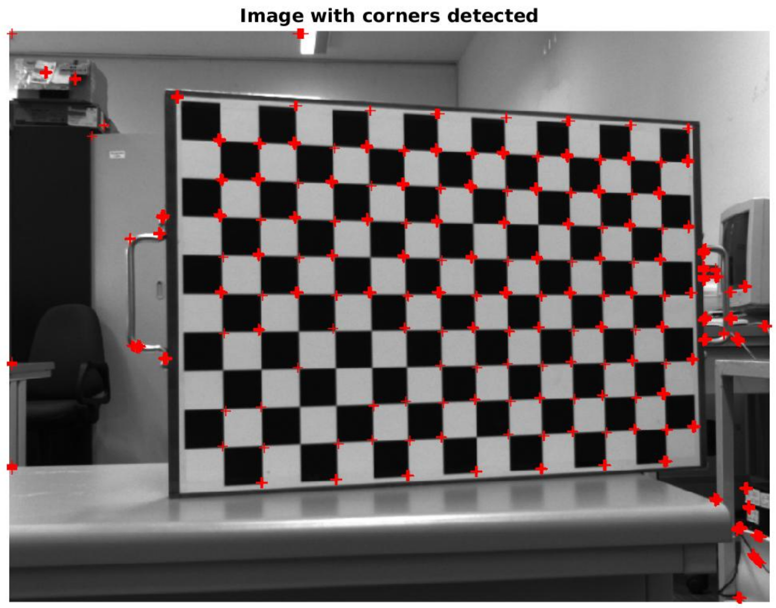

Figura 2: Imagem chess2 com cantos detectados, vizinhança 3x3

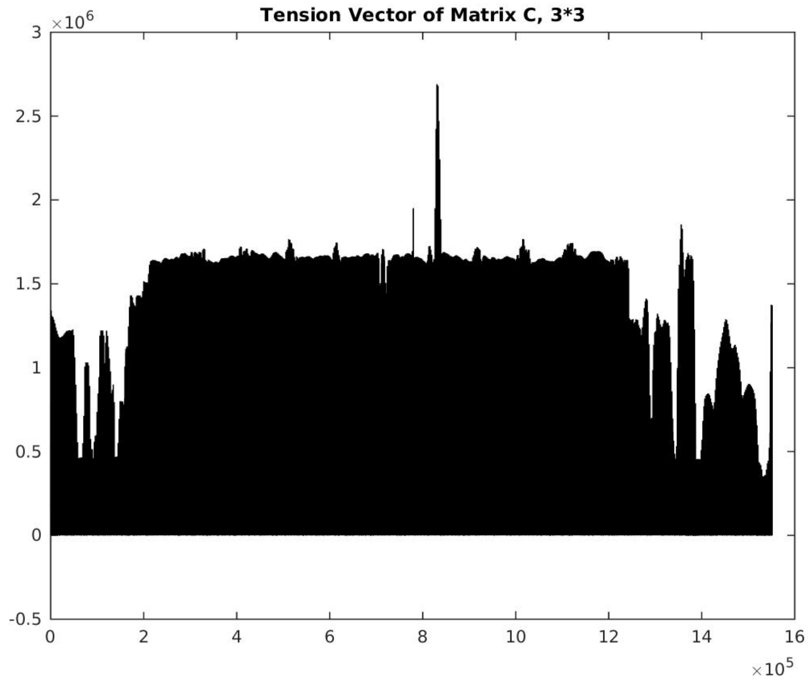

Figura 3: Tensor de estrutura da matriz C, vizinhança 3x3

Como é possível verificar, houve uma boa detecção dos cantos, com valor de σ a 250000.

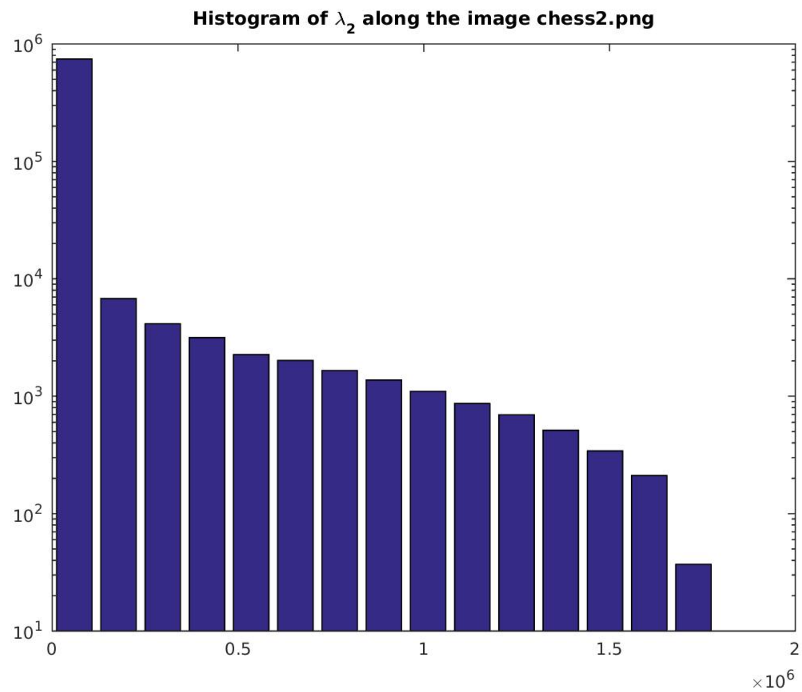

Figura 4: Histograma de λ_2 da imagem chess2, vizinhança 5x5

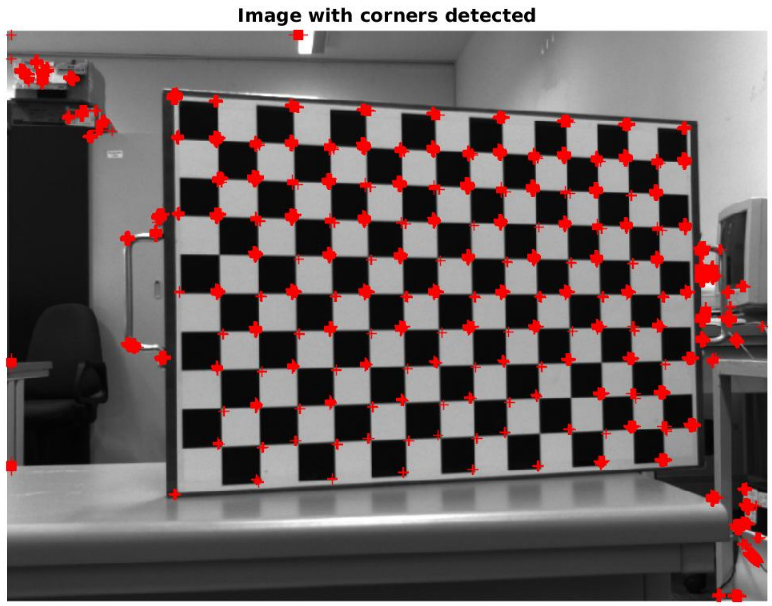

Figura 5: Imagem chess2 com cantos detectados, vizinhança 5x5

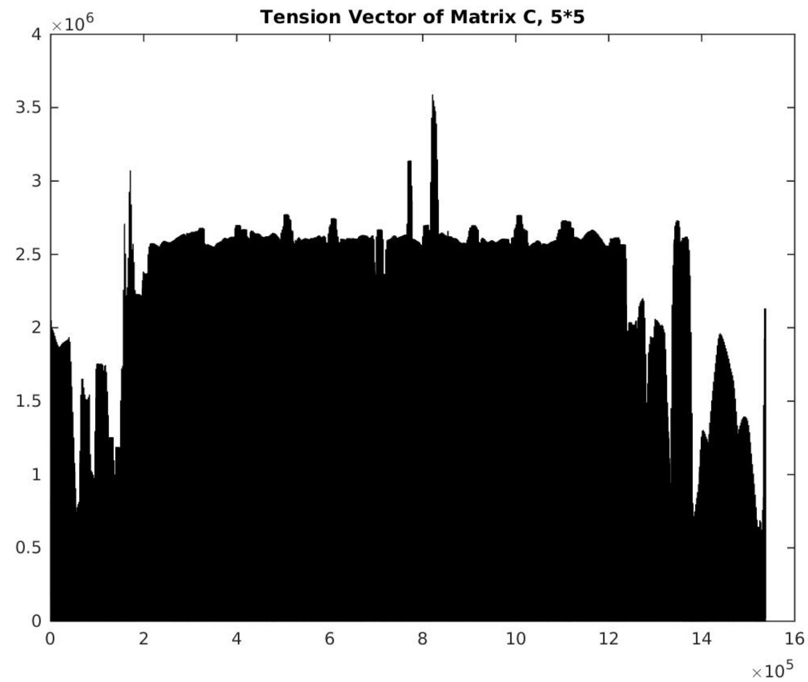

Figura 6: Tensor de estrutura da matriz C, vizinhança 5x5

Neste caso, também verifica-se uma boa detecção dos cantos, com bastantes cantos sobrepostos e valor de σ, 250000, tal como anteriormente.

Exercício 4

Agora, o algoritmo será testado numa imagem real, "corners.jpg", com vizinhanças de 3x3 e 5x5.

Irá ser verificado o seu histograma e a detecção dos cantos na imagem.

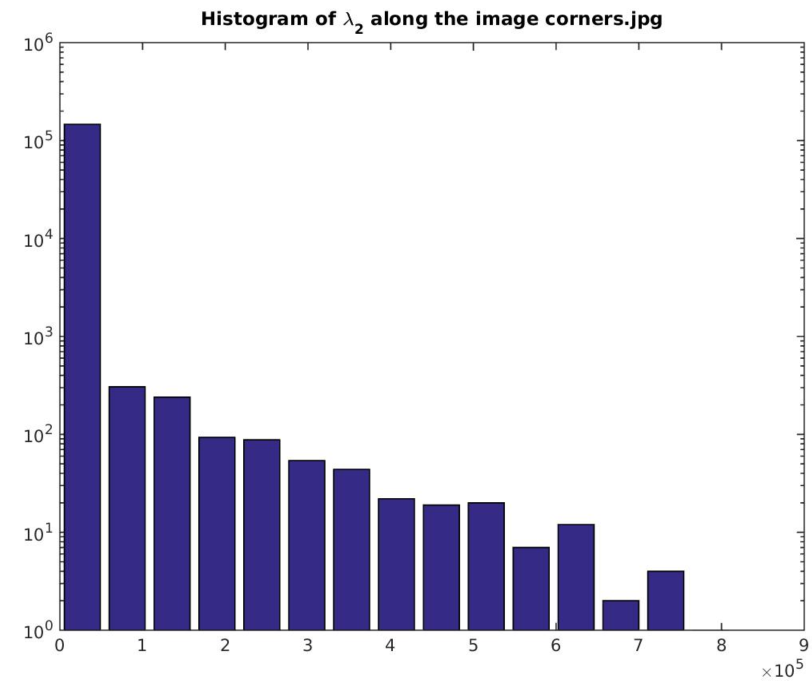

Figura 7: Histograma de λ_2 da imagem corners, vizinhança 3x3

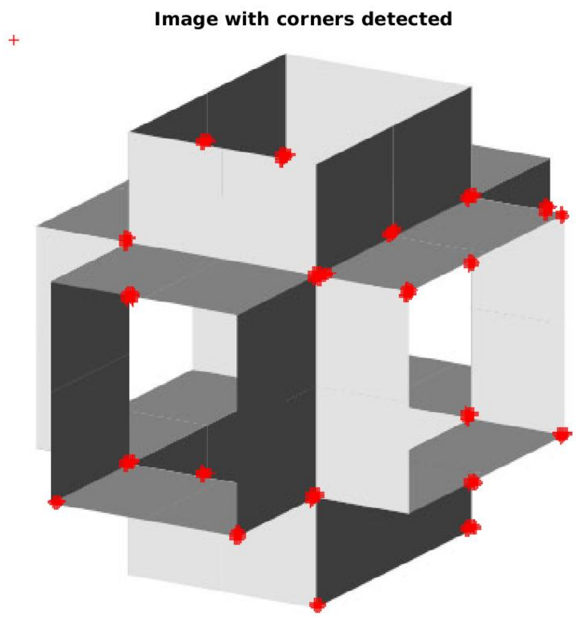

Figura 8: Imagem corners com cantos detectados, vizinhança 3x3

Como é possível verificar, houve uma boa detecção dos cantos, com valor de σ a 50000.

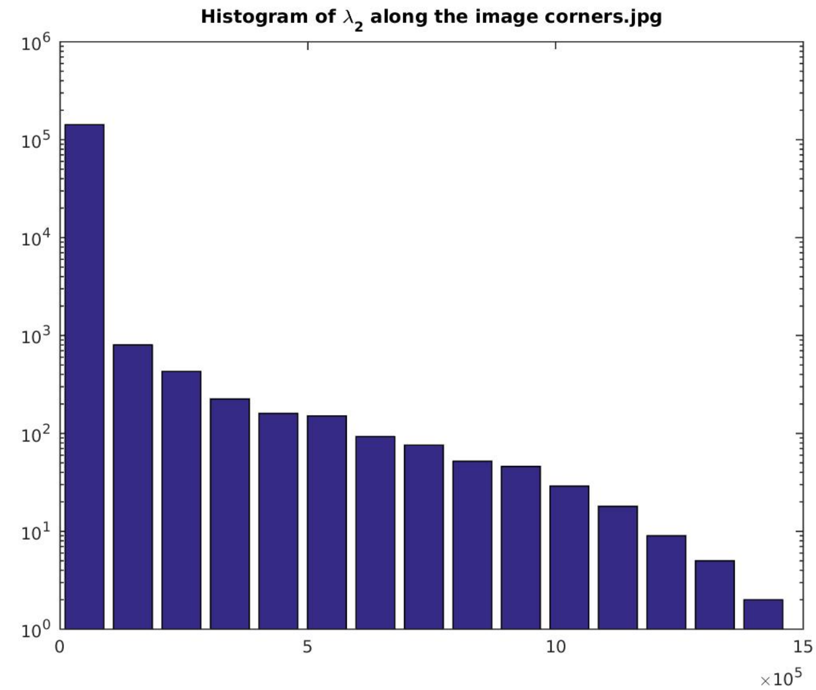

Figura 9: Histograma de λ_2 da imagem corners, vizinhança 5x5

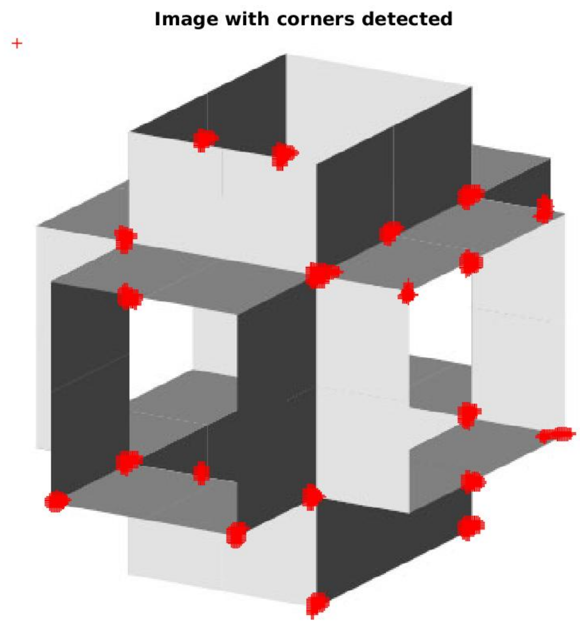

Figura 10: Imagem corners com cantos detectados, vizinhança 5x5

Neste caso, também verifica-se uma boa detecção dos cantos, com bastantes cantos sobrepostos e valor de σ, 50000, tal como anteriormente.

Parte dois

Nesta parte, serão mostrados os resultados derivados da implementação da transformada de Hough sobre rectas e sobre circunferências.

Para podermos fazer o display das matrizes que seram usadas para a detecção das rectas e das circunferências usámos a função houghTransform.m onde começamos por definir o espaço de Hough, seguindo-se da detecção das bordas da imagem, após isto criámos um acumulador e fizémos o display das matrizes, onde resultou nas seguintes figuras:

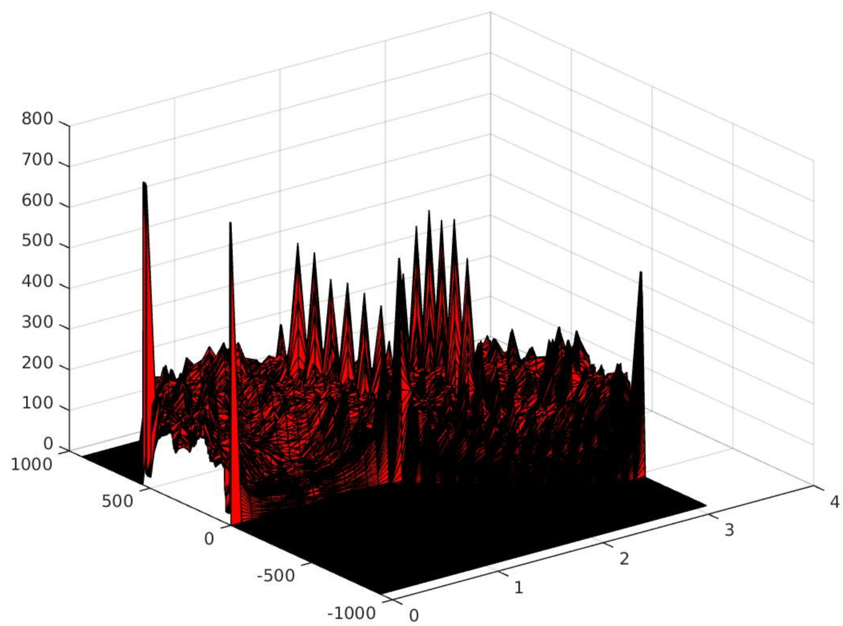

Figura 11: Gráfico do acumulador da transformada de Hough

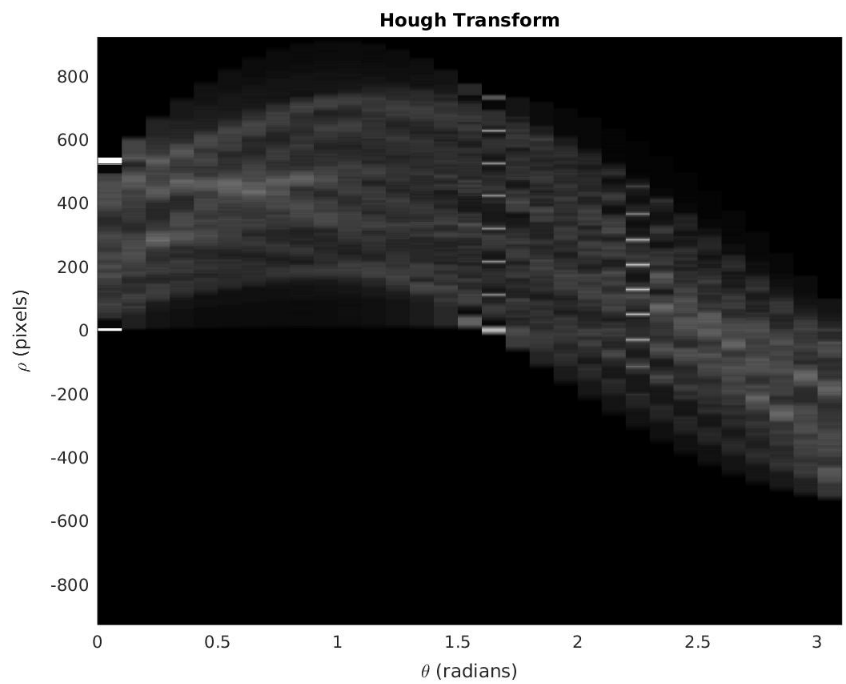

Figura 12: Gráfico do espaço de Hough

5.1 Hough Transform Lines

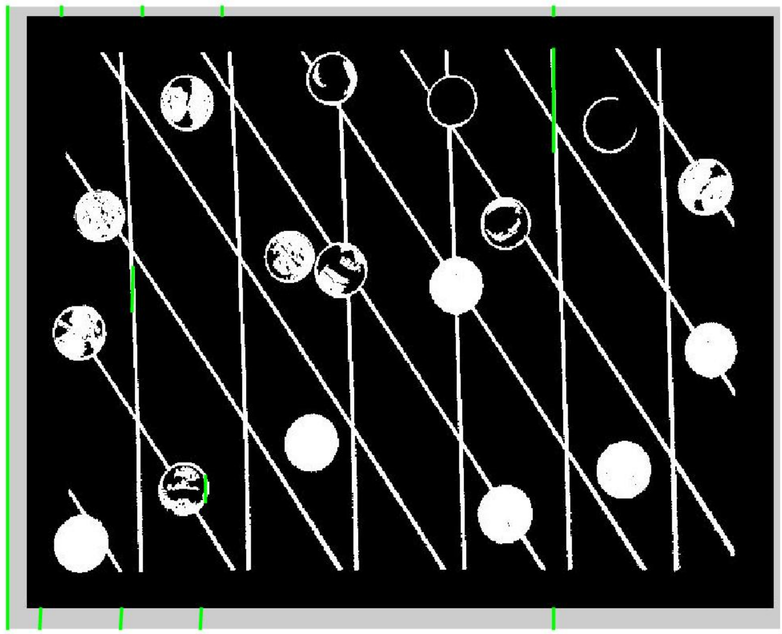

Figura 13: Intersecção da imagem com a detecção das linhas

Como é possível verificar, a detecção das rectas tem bastantes falhas e só detecta algumas partes, provavelmente devido à intensidade das próprias na imagem original.

5.2 Hough Transform Circles

Após a detecção das rectas, foi a vez de analisarmos a imagem tendo em conta as circunferências e usando mais uma vez a transformada de Hough. Após a execução da função circlesDetection.m obtivemos a seguinte imagem:

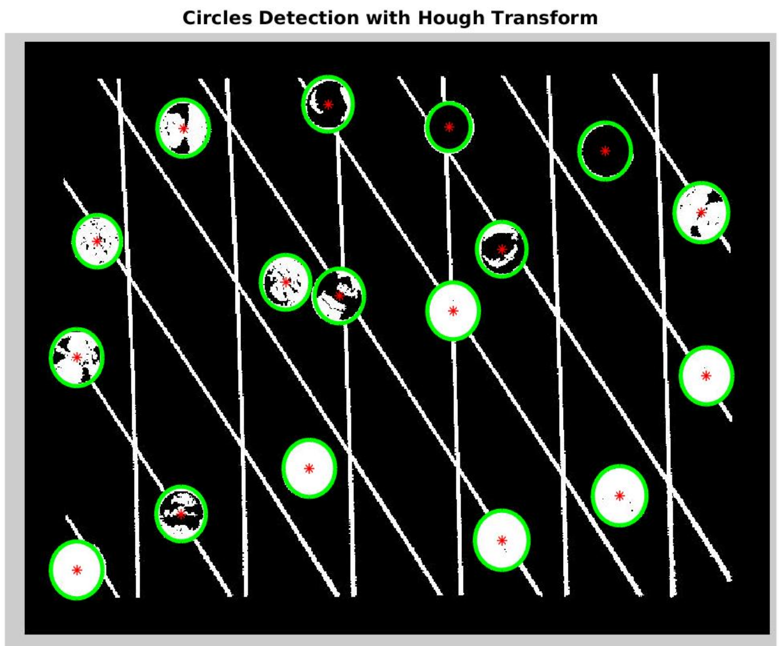

Figura 14: Intersecção da imagem com a detecção das circunferências

Como podemos observar pela figura, onde os centros das circunferências estão a vermelho e as margens das circunferências estão a verde, a aplicação da transformada de Hough foi bem sucedida aqui, uma vez que foram detectadas as circunferências correctamente.