



UNIVERSIDADE FEDERAL DE SANTA CATARINA  
CAMPUS BLUMENAU  
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Leonardo dos Santos Schmitt

**Laboratório 2:** Visão Computacional em Robótica

## 1 ATIVIDADE 2

A Atividade 2 consiste no desenvolvimento de um sistema de visão computacional capaz de realizar o reconhecimento óptico de caracteres (OCR - *optical character recognition*) em imagens de etiquetas, no qual uma dessas imagens pode ser visualizada na Figura 1. Ao final, deve-se apresentar no terminal o texto contido na imagem, conforme ilustrado no Quadro 1.

Figura 1 – Imagem de uma etiqueta.



Fonte: Dr.Prof.Marcos Matsuo, 2024.

Quadro 1 – Saída esperada após identificação de caracteres.

1	ORIGEM SETOR D
2	
3	DESTINO SETOR E
4	
5	PRODUTO TELHA

Fonte: Autor, 2024.

As imagens com outras etiquetas, assim como um *template* contendo as letras do alfabeto, foram fornecidas pelo Dr. Prof. Marcos Matsuo. O *template* das letras, disponibilizado na imagem `templates_letras.png`, será utilizado em conjunto com as demais imagens para o reconhecimento de caracteres por meio de *template matching*. Ressalta-se que as imagens possuem diferentes orientações, o que pode influenciar a identificação dos caracteres. Sendo assim, é necessário realizar uma homografia planar para alinhar todas as imagens em uma única orientação e, após isso, realizar a identificação dos caracteres.

### 1.1 SOLUÇÃO DA ATIVIDADE 1

Conforme elucidado anteriormente, esta atividade solicita o desenvolvimento de um sistema que realiza o reconhecimento de caracteres em imagens contendo diferentes

etiquetas. Esta seção apresentará a solução desenvolvida pelo autor, a qual abrange um contexto geral do processo, destacando detalhadamente cada fase e como ela pode ser implementada. Para fins práticos e de entendimento, esse tópico será dividido em subseções, contudo, todos eles serão conectadas.

### 1.1.1 Ajuste das Imagens

De primeiro momento, é necessário realizar uma identificação prévia das imagens em relação à sua orientação, pois, conforme mostrado na Figura 2, há duas imagens que possuem a mesma etiqueta, mas com perspectivas diferentes. Para desenvolver um algoritmo genérico que identifique todas as imagens, deve-se realizar uma homografia planar, ou transformação de perspectiva.



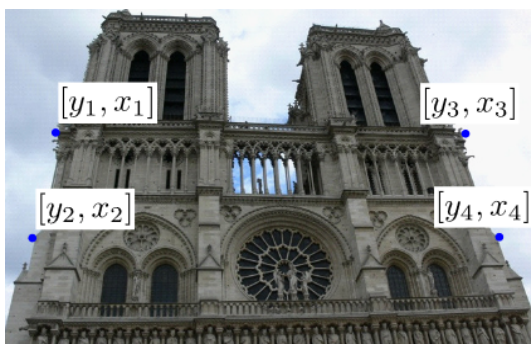
(a) Primeira perspectiva.



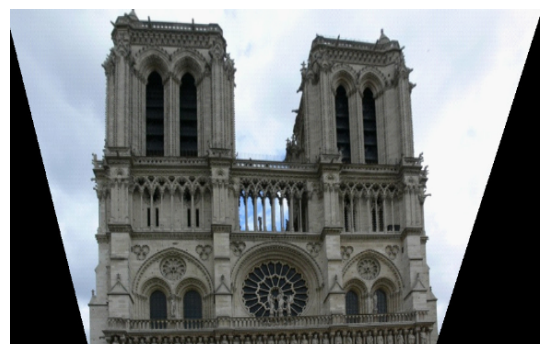
(b) Segunda perspectiva.

Figura 2 – Imagens com etiquetas iguais e perspectivas diferentes.

Este método consiste na relação entre dois planos em diferente perspectivas, mantendo a relação de pontos, linhas e formas. Um exemplo pode ser visto na Figura 3, no qual a Figura 3a, representa um foto tirada mais abaixo, fazendo com que o objeto em questão da cena tenha a sua base maior que a parte superior, contudo, dependendo da aplicação essa perspectiva pode não ser a mais correta de se analisar a imagem. Sendo assim é necessário realizar uma transformação para que o objeto fique de acordo com sua parte superior, conforme mostrado na Figura 3b.



(a) Imagem sem transformação de perspectiva.



(b) Imagem com transformação de perspectiva.

Figura 3 – Transformação de perspectiva entre imagens.

Em vista disso, pode-se utilizar a relação entre as posições dos *pixels* da imagem original e da transformada. Ou seja, deve-se selecionar as coordenadas na imagem original que serão mapeadas para as novas coordenadas da imagem transformada, que também pode ser definida. Após isso, cria-se uma matriz de transformação de homografia, conforme apresentado na Equação (1), onde, ao alterar os valores dos coeficientes  $h_{ij}$ , obtêm-se diferentes transformações de perspectiva.

$$\begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{bmatrix} = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,1} & h_{3,2} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \quad (1)$$

É importante ressaltar que na matriz de transformação de homografia, existem 8 coeficientes que devem ser determinados. Sendo assim, é necessário definir pelo menos quatro pares de pontos na imagem original e na desejada, conforme mostrado na Figura 3a, onde foram selecionados 4 pontos na imagem original. Estes por sua vez, podem ser relacionados à Equação (1), gerando assim, uma nova representação, conforme mostrado nas equações abaixo (Equação (2) e Equação (3)).

$$x'_i = \frac{\tilde{x}_i}{\tilde{z}_i} = \frac{h_{1,1}x_i + h_{1,2}y_i + h_{1,3}}{h_{3,1}x_i + h_{3,2}y_i + 1}. \quad (2)$$

$$y'_i = \frac{\tilde{y}_i}{\tilde{z}_i} = \frac{h_{2,1}x_i + h_{2,2}y_i + h_{2,3}}{h_{3,1}x_i + h_{3,2}y_i + 1}. \quad (3)$$

As equações acima podem ser reescritas de forma a obter um sistema de equações lineares, resultando em uma matriz  $8 \times 8$  denominada  $A$  e em vetores coluna  $h$  e  $b$ , correspondendo, respectivamente, ao vetor de incógnitas e ao vetor de variáveis independentes, conforme apresentado na Equação (4). Para realizar a implementação do algoritmo, o autor criou uma função chamada **homografia()**, que está contida no módulo **visaoComputacional** criado em sala de aula. A função utiliza um laço de repetição para obter os valores de todas as incógnitas e retorná-los para uma variável, que neste trabalho foi denominada de  $H$ .

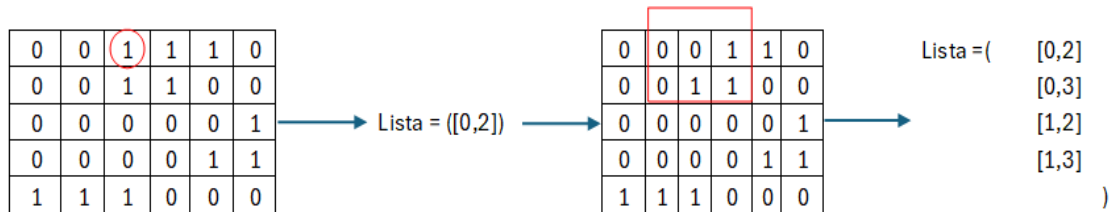
$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1x_1 & -x'_1y_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1x_1 & -y'_1y_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x'_2x_2 & -x'_2y_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y'_2x_2 & -y'_2y_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x'_3x_3 & -x'_3y_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -y'_3x_3 & -y'_3y_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x'_4x_4 & -x'_4y_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -y'_4x_4 & -y'_4y_4 \end{bmatrix} \begin{bmatrix} h_{1,1} \\ h_{1,2} \\ h_{1,3} \\ h_{2,1} \\ h_{2,2} \\ h_{2,3} \\ h_{3,1} \\ h_{3,2} \end{bmatrix} = \begin{bmatrix} x'_1 \\ y'_1 \\ x'_2 \\ y'_2 \\ x'_3 \\ y'_3 \\ x'_4 \\ y'_4 \end{bmatrix}. \quad (4)$$

No caso da aplicação da homografia planar para este trabalho, os quatro pontos utilizados da imagem original são os pixels que representam as coordenadas de cada forma em cor verde, conforme visto na Figura 1. Esses pontos serão mapeados para as coordenadas  $[0, 0]$ ,  $[0, n\_linhas-1]$ ,  $[n\_colunas-1, n\_linhas-1]$ ,  $[n\_colunas-1, 0]$  da nova imagem, onde as variáveis `n_linhas` e `n_colunas` representam o número de linhas e colunas definidas pelo autor, que neste caso, foram 331 e 666, respectivamente.

Após o conceito de homografia definido, antes de aplicá-la, é necessário utilizar um recurso importante, que é a análise ou rotulagem de componentes conectados. Essa etapa pode ser realizada com a função `connectedComponents()` da biblioteca *OpenCV*, que identifica e retorna regiões de interesse em uma imagem binária.

O algoritmo básico percorre a imagem binária até encontrar um *pixel* branco. Ao encontrá-lo, suas coordenadas são adicionadas a uma lista e o *pixel* é removido da imagem binária. Para tornar o processo mais eficiente, é possível analisar a vizinhança ao redor desse *pixel* branco, caso haja outros *pixels* brancos nas proximidades, eles também são adicionados à lista e removidos em seguida, conforme apresentado na Figura 4. Cada componente identificado é enumerado de acordo com sua posição na lista, começando a partir do índice 0.

Figura 4 – Parte do algoritmo básico para rotulagem de componentes conectados.



Fonte: Autor, 2024.

Com os componentes identificados na imagem, é possível extrair algumas características de cada um, como: centroide, área e os pontos que definem a *bounding box* (caixa delimitadora) do componente. Essas características são calculadas através dos momentos, destacando-se os momentos de ordem zero e um. O momento de ordem zero é obtido pela soma de todos os *pixels* da região, representando assim a área, conforme apresentado na Equação (5). Já os momentos de ordem um são calculados com base na posição dos *pixels* nas coordenadas  $x$  e  $y$  em relação à origem, permitindo determinar o centro de massa, conforme apresentados na Equação (6) e Equação (7), respectivamente. Além disso, os pontos que definem a *bounding box* do componente são obtidos a partir das coordenadas mínimas e máximas dos *pixels* que pertencem a esse componente. Todos

esses cálculos foram adicionas em uma função chamada `analisaRegioes()`, continda no módulo `visaoComputacional`, que retorna um dicionario com os pontos, centroide e área.

$$M_{00} = \sum_x \sum_y I(x, y) \quad (5)$$

$$M_{10} = \sum_x \sum_y x \cdot I(x, y) \quad (6)$$

$$M_{01} = \sum_x \sum_y y \cdot I(x, y) \quad (7)$$

Com as características necessárias definidas, no caso deste trabalho, pode-se utilizar as coordenadas dos centroides para realizar a homografia planar. Sendo assim é necessário o armazenamento da imagem de uma etiqueta em uma variável **I**, que será uma matriz. Após isso, deve-se converter a imagem para o espaço de cor binário, para que seja possível rotular os componentes conectados e extrair informações sobre as regiões.

Como as imagens disponibilizadas estão no formato BGR, deve-se utilizar a segmentação de cor, convertendo a cor do *pixel* para o espaço de cor binário. Para realizar tal procedimento, deve-se definir as cores de referência do objeto, que podem ser representadas pelas componentes *br*, *gr* e *rr*, sendo azul, verde e vermelho, respectivamente. Essas componentes, quando aplicadas em código, são dispostas em um vetor e armazenadas em uma variável, como por exemplo `cor_referencia = [b_r, g_r, r_r]`, sendo que cada cor pode ser referenciada pelo seu respectivo índice na variável armazenada.

Neste trabalho, a cor de referência utilizada pelo autor foi [120, 180, 106]. Ao definir as cores de referência, é possível realizar a segmentação de cor. Existem funções já prontas que conseguem realizar este processo, entretanto, o método utilizado neste caso, é a distância euclidiana, que verifica a similaridade de cores através dos *pixels* mais semelhantes e suas distâncias. Ou seja, os *pixels* com cores mais semelhantes, no cálculo da distância entre **I** e as cores de referência, terão uma distância euclidiana menor.

A distância euclidiana pode ser armazenada em uma variável **D**, que irá representar uma imagem, onde cada *pixel* **D**[y, x] é dado pela distância euclidiana entre o *pixel* **I**[y, x], da imagem de entrada, e a cor de referência. Na Equação (8), é possível visualizar o cálculo em questão, onde **B**[y, x], **G**[y, x] e **R**[y, x] referem-se aos *pixels* da camada BGR da imagem original.

$$\mathbf{D}[\mathbf{y}, \mathbf{x}] = \sqrt{(\mathbf{B}[\mathbf{y}, \mathbf{x}] - b_r)^2 + (\mathbf{G}[\mathbf{y}, \mathbf{x}] - g_r)^2 + (\mathbf{R}[\mathbf{y}, \mathbf{x}] - r_r)^2}. \quad (8)$$

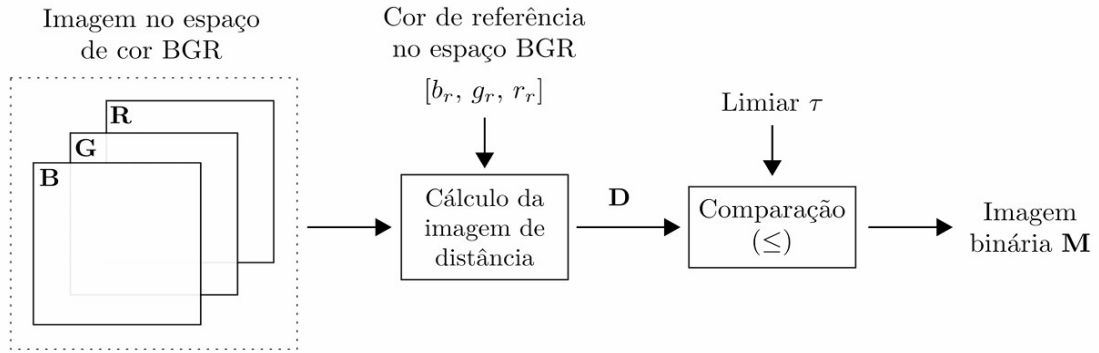
Com a nova imagem **D** obtida, é possível realizar a limiarização dos *pixels* da mesma com o limiar  $\tau$  definido pelo projetista ( que para este caso foi 50), sendo que o resultado será armazenado em uma imagem binária **M**, conforme apresenta a Figura 5,

que ilustra o fluxo que o algoritmo deve seguir. Em outras palavras, um limiar definido por meio de experimentação estabelece os limites em que o *pixel* encontra-se. Se o valor do pixel  $D[y, x]$  for menor ou igual ao valor do limiar, o *pixel* resultante na imagem  $M$  será 255, representando a cor branca no espaço de cor binário com valor `uint8`. Caso contrário, o *pixel* será 0, representando a cor preta nesse mesmo espaço.

A Equação (9) representa o que foi brevemente mencionado. Além disso, esse tipo de implementação, foi dado através da função `np.where()` do modulo `numpy`, no qual deve-se passar a condição e os valores de mapeamento para a mesma, no caso 255 e 0.

$$M[y, x] = \begin{cases} 255, & \text{se } D[y, x] \leq \tau \\ 0, & \text{caso contrário} \end{cases} \quad (9)$$

Figura 5 – Fluxo a ser seguindo pelo Algoritmo de segmentação baseada em cor e em medida de distância.



Fonte: Dr.Prof.Marcos Matsuo, 2024.

Com a segmentação de cor realizada, deve-se extrair apenas os objetos desejados da imagem binária, conforme apresentado na Figura 8, que ilustra a segmentação de cor da Figura 1. Já com os elementos segmentados, pode-se utilizar a função `analisaRegioes()` para calcular o valor do centroide de cada componente. Após obter os valores dos centroides, esses serão os quatro *pixels* selecionados da imagem original que devem ser mapeados para as novas coordenadas da imagem transformada.

Através da Figura 10 é possível visualizar que a mesma foi redimensionada. A princípio, pode não parecer clara a diferença, no entanto, ao considerar que a imagem atende escalas diferentes, conclui-se que a homografia planar foi realizada com sucesso. Além do redimensionamento, outra ação que pode ser realizada é a remoção das bordas, que o autor implementou para fins práticos, facilitando a identificação de caracteres, como será demonstrado adiante.

Esse procedimento utiliza a reconstrução morfológica, que envolve três imagens: a imagem de entrada, uma máscara (cópia da imagem de entrada) e uma imagem de

marcadores. Nessa imagem de marcadores, as bordas são destacadas e configuradas com `Marker[1:-1, 1:-1] = 0`. Em seguida, a mesma passa por uma dilatação com um *kernel* de 3x3 e, depois, é aplicada uma interseção lógica E (AND) entre a máscara e os marcadores, utilizando a função `bitwise_and()`.

Esse ciclo de dilatação e interseção se repete até que não haja mais mudanças nos marcadores. Por fim, a imagem resultante desse processo é subtraída da imagem original, removendo as bordas e gerando uma nova imagem, conforme ilustrado na Equação (10), onde  $R_G(F)$  representa a operação de erosão e a interseção. Na Figura 12, que se encontra na próxima seção, é possível visualizar a imagem com e sem as bordas.

$$H = I - R_g(F), \quad (10)$$

### 1.1.2 Criação dos *Templates*

Após a realização da homografia planar das imagens, o segundo passo é obter os *templates* que serão utilizados no algoritmo de *template matching*. Para isso, pode-se aplicar os mesmos conceitos de componentes conectados abordados na subseção anterior, em conjunto com a imagem `templates_letras.png`, apresentada abaixo (Figura 6). Primeiro, realiza-se a conversão da imagem para escala de cinza, seguida da aplicação de um *threshold* utilizando o limiar de *Otsu* para converter a imagem para o espaço de cor binário. Dessa forma, as letras ficam com os valores de *pixels* iguais a 255 e o fundo com os valores em 0.

Ao obter os componentes conectados, eles serão extraídos de forma desordenada, ou seja, não seguirão a ordem do alfabeto conforme apresentado na Figura 6. Para corrigir isso, utilizou-se a função `sorted()` do *Python*, com base na coordenada x do centroide de cada componente (letra).

Figura 6 – Imagem contendo todas as letras do alfabeto para criação dos *templates*.

ABCDEFGHIJKLMNOPQRSTUVWXYZ

Fonte: Dr.Prof.Marcos Matsuo, 2024.

Com as letras ordenadas, foi criado um dicionário vazio (intitulado como `dic = {}`) e uma lista contendo todas as letras do alfabeto. Isso é necessário para formar pares de chave e valor, onde cada chave é um caractere da lista (por exemplo, a letra 'A') e o valor corresponde às coordenadas extremas de cada pixel da imagem, que são usadas para gerar o *template* extraído da função `inforegioes()`, responsável pela análise de componentes conectados. Os pontos relevantes são obtidos através de `p1 =`



`infoRegioes[i]['bb_point1']` e `p2 = infoRegioes[i]['bb_point2']`, onde essas variáveis contêm dois valores. Esses valores são utilizados para recortar a imagem, formando o *template* da seguinte forma: `template[p1[1]:p2[1], p1[0]:p2[0]]`. Isso indica que o *template* é extraído do ponto *p1* nas posições zero e um até o ponto *p2* nas mesmas posições.

Após obter os valores, um laço de repetição percorre a lista, pegando cada letra e adicionando-a ao dicionário. O índice correspondente à posição da letra é utilizado em `infoRegioes()` para coletar os pontos da letra desejada, permitindo assim realizar o recorte no *template* e armazená-lo no dicionário na posição da chave. Dessa forma, o dicionário pode ter uma estrutura como: `'A' = template_letra_a`.

### 1.1.3 Aplicação do *Template Matching*

Após a aplicação da homografia planar, a remoção das bordas e a criação dos *template*, é possível realizar o *template matching* por meio do cálculo da similaridade. Para isso, utilizamos dois laços de repetição aninhados. O laço externo percorre o dicionário, obtendo as letras que são as chaves e as imagens que são os valores correspondentes. Já o laço interno itera sobre a `infoRegiao` da imagem, acessando o índice atual do laço e a região, que representa o componente conectado.

Entretanto, para realizar a comparação entre as duas imagens, é necessário que tanto o *template*, quanto o elemento conectado da imagem, sejam do mesmo tamanho. Para isso, utilizou-se a função `resize()`, passando como parâmetros o *template* ou o elemento conectado, além da largura e altura desejadas. Neste caso, optou-se por um redimensionamento de  $50 \times 50$  *pixels*, tanto para o *template* quanto para o elemento da região.

Destaca-se que um dos problemas enfrentados pelo autor foi que, ao aplicar o *resize* na letra *I*, o *template* gerava correspondência em pontos existentes na imagem, conforme apresentado na Figura 7. Isso ocorreu porque, após o redimensionamento, tanto os pontos quanto a letra *I* tornavam-se semelhantes. Para corrigir esse problema, o autor utilizou o cálculo da área, mostrado na Equação (11), para estabelecer um *threshold* antes de calcular a similaridade, conforme apresentado no condicional da Equação (12). Nesse caso, o *threshold* foi definido como 150 para excluir imagens de pontos.

$$\text{Área} = \text{largura} \times \text{altura} \quad (11)$$

$$\text{Resultado} = \begin{cases} \text{Processar a imagem,} & \text{se área} > 150 \\ \text{Ignorar a imagem,} & \text{caso contrário} \end{cases} \quad (12)$$

Quando as duas imagens estão do mesmo tamanho, é possível calcular a similaridade entre elas. Nesse contexto, ressalta-se que o autor optou por não utilizar métodos

Figura 7 – Destaque dos pontos que são confundidos com a letra I.



Fonte: Autor, 2024.

mais simples, como o *SAD*, apresentado na Equação (13). Esse método resultava em similaridades muito baixas, mesmo após a aplicação de um fator multiplicativo, o que impedia a separação dos caracteres. Por isso, o autor optou pelo método *ZSSD*, que calcula a média de cada imagem (*template* e elemento conectado), subtrai essa média da respectiva imagem, eleva ao quadrado e soma todos os valores, conforme apresentado na Equação (14).

$$\text{SAD}(I_1, I_2) = \sum_{x=1}^W \sum_{y=1}^H |I_1(x, y) - I_2(x, y)|. \quad (13)$$

$$\text{ZSSD}(I_1, I_2) = \sum_{x=1}^W \sum_{y=1}^H ((I_1(x, y) - \mu_{I_1}) - (I_2(x, y) - \mu_{I_2}))^2. \quad (14)$$

Conforme apontado no parágrafo anterior, utilizou-se um fator multiplicativo (valor de 1.5) em conjunto com a similaridade para facilitar a identificação de caracteres. Para isso, foi implementado um condicional que verifica se a similaridade está abaixo de um limite estipulado pelo autor, com base na iteração e análise do algoritmo. Se a similaridade estiver abaixo de 365, a letra proveniente do dicionário é armazenada em uma lista, enquanto as coordenadas  $x$  e  $y$  do centroide são armazenadas em outras duas listas diferentes, conforme apresentado na Equação (15). Essas coordenadas serão utilizadas para mostrar a palavra no terminal de saída.

$$\text{Classificação} = \begin{cases} \text{Adicionar nas listas,} & \text{se similaridade} < 365 \\ \text{Ignorar,} & \text{caso contrário} \end{cases} \quad (15)$$

### 1.1.4 Apresentação da Mesagem

Por fim, para agrupar as letras em linhas de texto, primeiramente são definidas tolerâncias para o agrupamento vertical e horizontal. A tolerância vertical, definida como `tolerancia_y = 65` determina o intervalo em *pixels* para considerar que letras estão no mesmo nível. A tolerância horizontal, definida como `tolerancia_x = 50` define a distância mínima entre letras para que se insira espaços entre elas.

Em seguida, as letras e suas coordenadas são combinadas em uma lista de tuplas usando a função `zip()`, que associa cada letra às suas respectivas coordenadas  $x$  e  $y$ . Essa lista é então ordenada, primeiro pelo nível vertical e, dentro de cada nível, pela coordenada horizontal.

A partir disso, inicia-se a iteração pelas letras ordenadas. Para cada letra, verifica-se se ela está no mesmo nível vertical da linha atual. Se estiver, adiciona-se a letra à linha de texto em formação. Caso a letra atual esteja distante da anterior, proporcionalmente a tolerância horizontal, insere-se espaços para manter a separação. Quando se encontra uma letra em um nível diferente, a linha atual é finalizada e a nova letra inicia uma nova linha de texto.

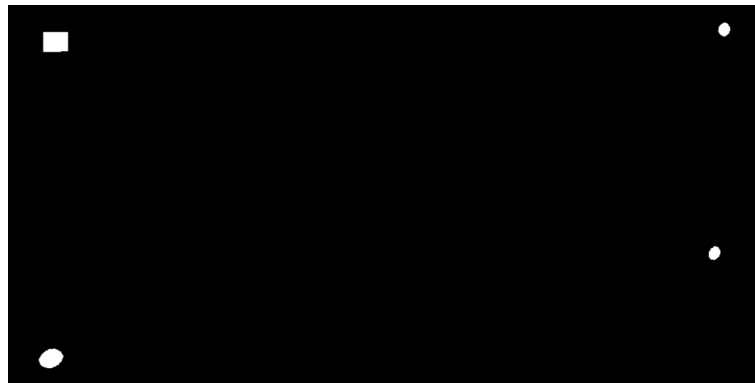
Ao final da iteração, a última linha de letras é adicionada ao texto formatado. Por fim, o resultado é impresso no terminal, apresentando o texto organizado conforme a estrutura original da imagem

## 1.2 RESULTADO OBTIDOS NA ATIVIDADE 1

Nesta seção, serão apresentados os resultados obtidos na Atividade 1, por meio de imagens. Entretanto, ressalta-se que não serão exibidas todas as 8 imagens, mas apenas dois tipos.

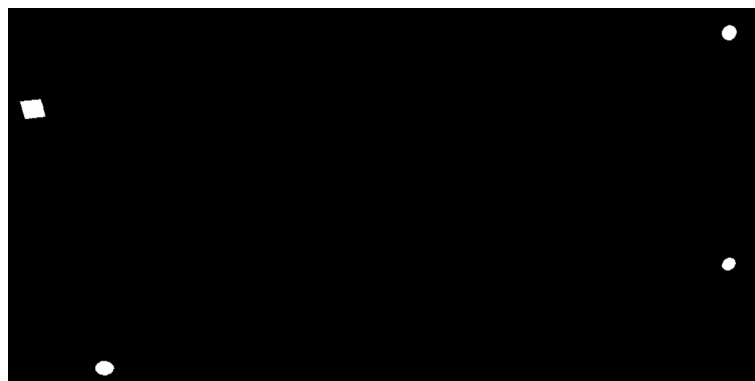
Primeiramente, apresenta-se a segmentação de cor da Figura 1, ilustrada pela Figura 8, onde os pontos verdes são mapeados para o espaço de cor binário. Isso ocorre, conforme mencionado na seção anterior, para possibilitar a análise de componentes conectados e, assim, realizar a homografia planar. Já na Figura 9, é possível observar outro exemplo de segmentação, que está relacionado com a Figura 2b. Portanto, conclui-se que a segmentação de cor foi realizada com sucesso.

Figura 8 – Segmentação de cor para Figura 1.



Fonte: Autor, 2024.

Figura 9 – Segmentação de cor para Figura 2b.

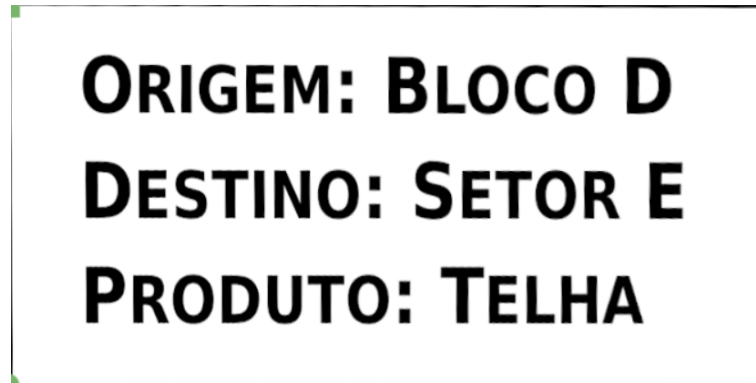


Fonte: Autor, 2024.

Após realizar a segmentação de cor, deve-se realizar a homografia planar, através dos centroides dos elementos conectados das imagens anteriores. Nesse caso, através da

Figura 11 e Figura 10, é possível verificar que as imagens, após a homografia planar, tiveram suas perspectivas alteradas. Novamente, ressalta-se que as duas representam a mudança de perspectiva da Figura 1 e Figura 2b, respectivamente.

Figura 10 – Nova perspectiva obtida para Figura 1



Fonte: Autor, 2024.

Figura 11 – Nova perspectiva obtida para Figura 2b



Fonte: Autor, 2024.

Após realizada a homografia planar e obtidas as novas perspectivas, pode-se partir para a remoção dos objetos que se encontram na borda, visíveis nas imagens acima. Neste caso, pode-se utilizar apenas a Figura 10 para realizar o processo de reconstrução morfológica abordado na seção anterior. Sendo assim, ao realizar o processo em questão, é possível passar da Figura 12a para a Figura 12b, no qual não se possui mais elementos na borda da imagem, e os únicos componentes conectados que retornam na `infoRegioes()` são os caracteres da etiqueta.

Dando sequência no algoritmo e nos resultados obtidos, com a remoção dos elementos na borda da imagem, deve-se obter os *templates* associados a cada letra. Nesse caso, como a imagem do *template* obtida é pequena, a Figura 13 é uma representação da

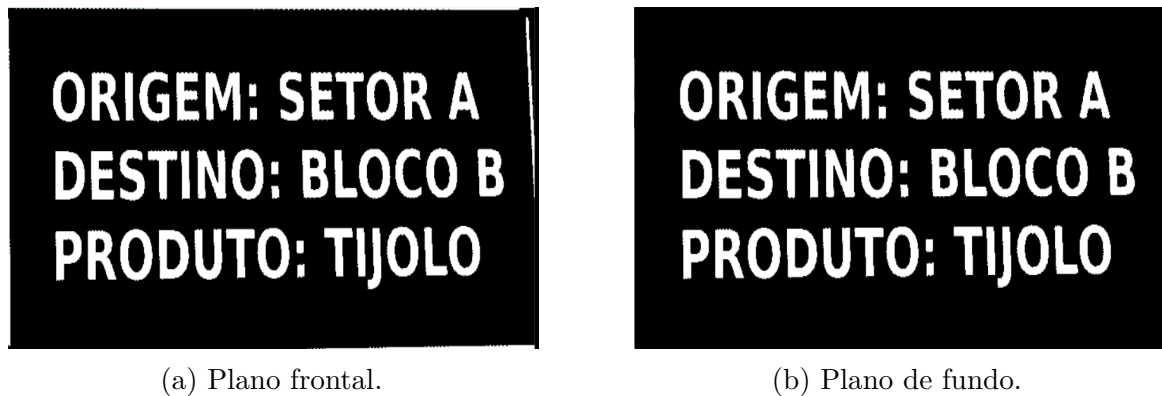


Figura 12 – Remoção dos elementos na borda da imagem.

Figura 6, no qual contém todas as letras do alfabeto, e logo abaixo está o *template* da letra *O*, mostrando que a criação do *template* funcionou.

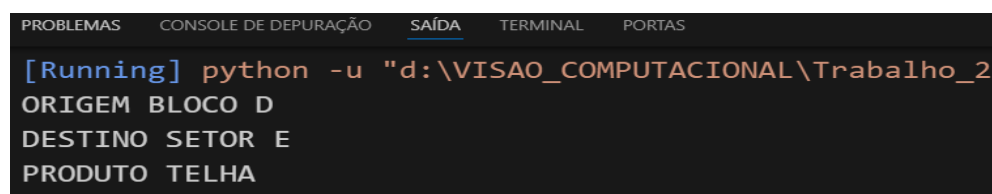
Figura 13 – Criação do *template* da letra *O* utilizando a Figura 6.



Fonte: Autor, 2024.

Por fim, em sequência da criação dos *templates*, da realização da homografia planar e a remoção dos elementos da borda, pode-se realizar o *template matching* entre as imagens. Na Figura 14, é possível visualizar a mensagem mostrada no terminal da etiqueta da Figura 1. Já na Figura 15, é possível visualizar a saída da imagem Figura 2b, concluindo assim que o código está funcional.

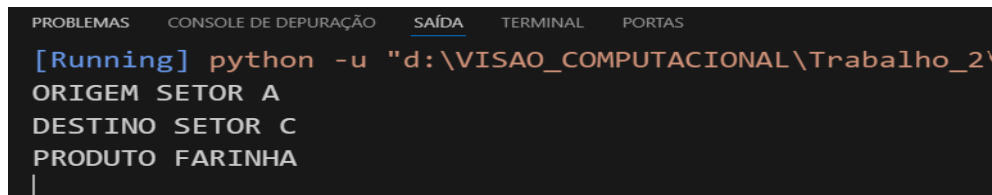
Figura 14 – Mensagem apresentada no terminal através da identificação do caracteres da Figura 1.



Fonte: Autor, 2024.

Destarte, conclui-se que o algoritmo de visão computacional para identificação de caracteres está funcional. Com ele, é possível identificar caracteres em imagens com etiquetas, mesmo quando estas apresentam diferentes perspectivas.

Figura 15 – Mensagem apresentada no terminal através da indexificação do caracteres da Figura 2b.



```
PROBLEMAS  CONSOLE DE DEPURAÇÃO  SAÍDA  TERMINAL  PORTAS
[Running] python -u "d:\VISA0_COMPUTACIONAL\Trabalho_2\'
ORIGEM SETOR A
DESTINO SETOR C
PRODUTO FARINHA
|
```

Fonte: Autor, 2024.