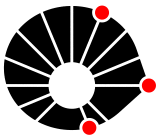
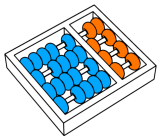




Detecção e Decodificação de Códigos de Barras em Imagens

L. F. R. Fonseca *H. Pedrini*

Relatório Técnico - IC-PFG-18-24
Projeto Final de Graduação
2018 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Detecção e Decodificação de Códigos de Barras em Imagens

Luiz Fernando Rodrigues da Fonseca Hélio Pedrini*

Resumo

Este relatório descreve as principais atividades desenvolvidas durante o Projeto Final de Graduação do curso de Engenharia de Computação do Instituto de Computação da Universidade Estadual de Campinas (UNICAMP). Neste trabalho, foram feitos estudos sobre detecção e decodificação de códigos de barras unidimensionais (1D) e bidimensionais (2D), com foco principal nos códigos EAN-13 e QR Code. Partindo do código em linguagem de programação Java da biblioteca ZXing, os algoritmos do EAN-13 e QR Code foram traduzidos para Python e, em seguida, foram realizados experimentos aplicando-se modificações nos métodos de binarização das imagens e adicionando pré processamento para realce de bordas. Os resultados dos diferentes métodos utilizados são comparados e discutidos, utilizando-se a acurácia de acerto na detecção e decodificação das informações como métrica.

1 Introdução

Códigos de barras são amplamente utilizados para se registrar informações e repassá-las com a leitura das unidades presentes nas imagens. Utilizações comuns são o registro e a leitura de preços e informações de produtos em supermercados, ou obter URLs (*Uniform Resource Locator*) de websites por meio da leitura do código. Estes códigos são divididos em duas categorias: códigos de barras unidimensionais (1D), como por exemplo o código EAN-13 (*European Article Number 13*), e códigos de barras bidimensionais (2D), como por exemplo o QR Code (*Quick Response*). Exemplos do código EAN-13 e QR Code são ilustrados na Figura 1.

O código EAN-13 é um padrão internacional composto por 13 dígitos, usado em transações globais para se identificar o tipo do produto, em uma configuração de pacote específica, de um fabricante específico. Já o QR Code foi criado em 1994 no Japão para ser utilizado na indústria automotiva, porém como possui vários tamanhos e pode armazenar até 4296 caracteres alfanuméricos, ganhou várias outras aplicações ao redor do mundo.

Para os códigos 1D, geralmente são utilizados leitores que emitem um raio laser vermelho que percorre todas as barras para se fazer a leitura das informações. Estes sensores geralmente são caros, o que torna a utilização de imagens uma ótima alternativa, já que praticamente todos os smartphones nos dias atuais possuem uma câmera. Em relação

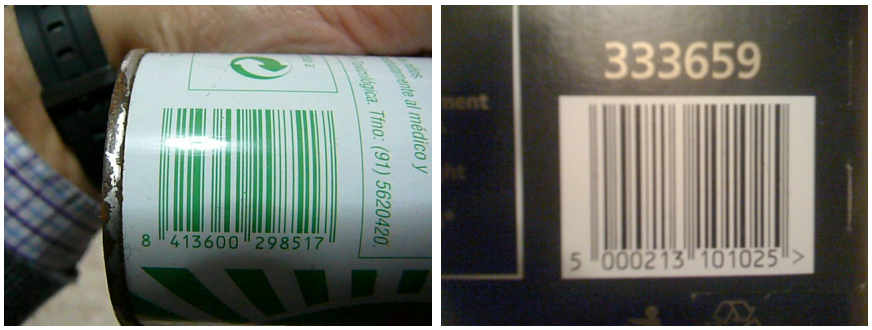
*Instituto de Computação, Universidade Estadual de Campinas, 13083-852 Campinas, SP.

(a) EAN-13 (b) QR Code

Figura 1: Exemplos do código EAN-13 e QR Code.

aos códigos 2D, as câmeras dos aparelhos celurares já possuem leitores para eles, e muitas aplicações já utilizam esta tecnologia, o que torna melhoras no processo de detecção e decodificação essenciais para a utilização destas aplicações.

Entre as dificuldades em se fazer a detecção e a leitura dos códigos, a binarização da imagem é um processo chave para que os algoritmos sejam capazes de obter corretamente as informações armazenadas. Esta etapa possui muitos desafios associados, pois imagens com pouca iluminação, ou com reflexão de raios de luz na região do código, ou mesmo borramento, são fatores que dificultam a leitura correta das informações. A Figura 2 ilustra esses efeitos.

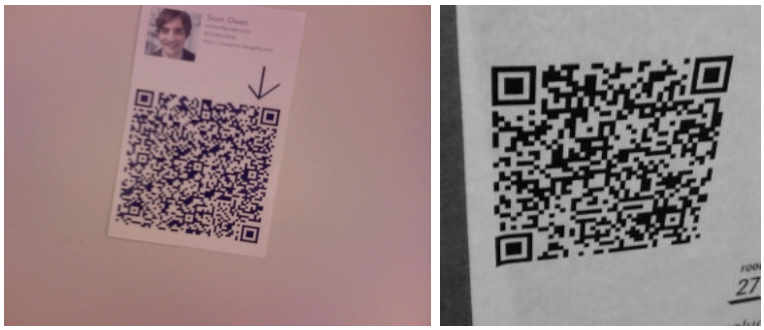


(a) Reflexão (b) Borramento

Figura 2: Imagens com reflexão de raios de luz e borramento.

Outro desafio está relacionado ao posicionamento e orientação do código de barras na imagem. Os algoritmos precisam ser robustos o suficiente para tratar casos em que o código não está alinhado horizontalmente, ou seja, casos em que possui um ângulo de rotação em relação ao eixo horizontal. Há também casos em que existe um efeito de distorção na imagem, fazendo com que o código não seja um retângulo ou um quadrado perfeito. A Figura 3 ilustra esses efeitos.

Neste relatório, a Seção 2 faz uma revisão de trabalhos anteriores e explica os métodos de detecção e decodificação dos códigos de barras 1D e 2D utilizados. A Seção 3 explica



(a) Rotação (b) Distorção

Figura 3: Imagens com rotação e distorção.

os métodos propostos para melhora do processo de decodificação. A Seção 4 mostra os resultados experimentais obtidos e a Seção 5 as conclusões e propostas de trabalhos futuros.

2 Conceitos e Trabalhos Relacionados

Há vários algoritmos para detecção de códigos de barras estudados nos últimos anos. Diferentes trabalhos utilizaram diferentes abordagens para a resolução do problema, como similaridade de histogramas para QR Code [2], *Aesthetic* QR Code que são aqueles que possuem imagens embutidas na região do código [4], detecção de códigos 1D com *Maximal Stable Extremal Region* [8], transformada de Hough [9], características locais [10] e Anaĺise de Componentes Principais [11].

Outros trabalhos utilizaram redes neurais e aprendizado de máquina profundo (*deep learning*) para detecção de códigos 1D, de forma que os algoritmos sejam invariantes quanto a ângulo de rotação, como [5] e [12]. Algumas abordagens também foram propostas para tratar códigos 1D que estejam borrados ou riscados [3], e QR Codes com efeito de luz e distorção geométrica, como em [6] e [7].

A maioria dos métodos citados, porém não todos, utiliza etapas comuns no início do algoritmo para se detectar os códigos nas imagens. Primeiro, a imagem é convertida para tons de cinza, e depois é aplicado um algoritmo de limiarização para binarizar a imagem. Este trabalho teve como foco estudar os códigos de barras de maneira geral, mas com um foco na etapa de pré processamento e binarização.

O trabalho se iniciou fazendo a tradução do código em Java da biblioteca ZXing [1] para Python 2, com o auxílio das bibliotecas OpenCV, NumPy e reedsolo, biblioteca que implementa o código detector e corretor de erros Reed-Solomon [14]. As imagens utilizadas para experimentos também são da biblioteca ZXing, sendo 157 imagens de códigos EAN-13 e 184 imagens de QR Code.

2.1 Algoritmos de Limiarização

Para se transformar uma imagem de tons de cinza para binária, são utilizados algoritmos de limiarização, que são divididos em dois grupos: global e local. A limiarização global utiliza um único limiar para a imagem toda, e os pixels da imagem são comparados com esse limiar para saber se devem possuir valor 0 ou valor 1. Já a limiariação local utiliza um limiar adaptativo, ou seja, cada pixel da imagem pode possuir um valor diferente de limiar, e diferentes estratégias podem ser utilizadas para se encontrar esse limiar local.

2.1.1 Limiarização Global Simples

Na limiarização global simples, é escolhido a priori um único valor de limiar T para se binarizar a imagem, sendo que pixels com valores menores do que T recebem o valor 1 e pixels com valores maiores do que T recebem o valor 0. Para os experimentos com EAN-13, o limiar escolhido foi 100, e para os experimentos com QR Code o limiar escolhido foi 110.

2.1.2 Limiarização Global de Otsu

O método de Otsu é uma evolução da limiarização global simples. Este algoritmo assume que o histograma da imagem é composto por duas classes, os pixels do objeto e os pixels do fundo. A ideia é então minimizar a variância dentro da mesma classe, escolhendo o valor do limiar T para isso. O método também possui a desvantagem de não possuir um bom desempenho caso o histograma das imagens não seja bimodal. Mais detalhes podem ser encontrados em [13].

2.1.3 Limiarização Local Adaptativa Média

Este método utiliza uma janela quadrada deslizante e calcula a média dentro desta janela como o valor do limiar para cada pixel da imagem. Também é possível utilizar uma constante para se subtrair do valor da média para ser de fato o limiar. Tanto para o código EAN-13 quanto para o QR Code, o tamanho da janela utilizado foi de 31, sem constante de subtração.

2.1.4 Limiarização Local Adaptativa Gaussiana

Este método utiliza uma janela quadrada deslizante e calcula uma soma ponderada por um filtro Gaussiano como valor do limiar para cada pixel da imagem. Também é possível utilizar a constante de subtração neste caso. O filtro Gaussiano é um vetor de tamanho $ksize \times 1$, sendo $ksize$ o tamanho do lado da janela quadrada, e suas equações são dadas a seguir:

$$G_i = \alpha * e^{-(i-(ksize-1)/2)^2/(2*sigma^2)}$$

em que $i = 0 \ldots ksize - 1$ e α é um fator de escala escolhido para que $\sum_i G_i = 1$. Para os experimentos do código EAN-13, o tamanho da janela utilizado foi 31 sem constante de subtração, mas no QR Code, o tamanho da janela utilizado foi 51 com constante de subtração 2.

2.1.5 Limiarização Local da Linha do ZXing

Este método é utilizado na biblioteca ZXing nos casos dos códigos 1D, pois considera apenas a informação de uma linha da imagem para os cálculos dos limiares. Desta forma, códigos que sofreram rotação em imagens com ângulo de 45 graus podem ser de mais difícil leitura por este método, mas que para rotações menores e em casos comuns é suficiente. A grande vantagem deste método é que ele opera por linha, então em casos onde o código está no centro da imagem não é necessário percorrer a imagem inteira.

Primeiramente, no método é calculado um histograma da linha da imagem com 32 bins. A partir deste histograma, é calculado o pico mais alto. Depois, é calculado um segundo pico que maximiza uma função de *score* para cada x no histograma, sendo *primeiro_pico* a posição do primeiro pico no histograma:

$$score_1 = hist[x] * (x - primeiro\_pico)^2$$

Com o pico mais alto e o segundo pico, é encontrado um vale entre estes picos que maximiza outra função de score para cada x no histograma, sendo *segundo_pico* a posição do segundo pico no histograma e *max_hist* o valor máximo do histograma:

$$score_2 = (x - primeiro\_pico)^2 * (segundo\_pico - x) * (max\_hist - hist[x])$$

O melhor valor do segundo *score* é multiplicado por 8 para se ter o valor de um limiar, chamado de ponto preto. Com este valor, para cada pixel da linha é utilizado um filtro $[-1, 4 - 1]$ com peso 2, e a soma ponderada é comparada com o ponto preto para se saber se o pixel terá valor 1 ou 0.

2.1.6 Limiarização Local Total do ZXing

Este método é utilizado na biblioteca ZXing para os códigos 2D, e limiariza toda a imagem. Neste algoritmo, a imagem é dividida em blocos de tamanho 8×8 pixels e, para cada bloco, é calculado um valor de ponto preto.

Para cada bloco, são calculados a soma dos valores dos pixels do bloco, o valor mínimo e o valor máximo. É então calculada a média com a soma. Se a diferença entre o máximo e o mínimo for maior que uma constante, neste caso 24, é utilizada a média como ponto preto do bloco. Caso contrário, significa que a variância dentro do bloco é baixa.

Para o segundo caso, significa que o bloco é uma área somente com pixels pretos ou somente com pixels brancos, e mais processamento precisa ser feito. É então calculada uma média com pontos pretos próximos previamente calculados, sendo *bps* os pontos pretos:

$$media\_vizinhanca = (bps[y - 1][x] + (2 * bps[y][x - 1]) + bps[y - 1][x - 1])/4$$

Se o valor mínimo for menor do que esta média da vizinhança, ela é utilizada como ponto preto, senão, é utilizado o valor mínimo dividido por 2.

Com os pontos pretos calculados para cada bloco, percorre-se todos os blocos calculando uma média para cada um considerando uma grade de 5×5 nos pontos pretos vizinhos. Por fim, utiliza-se esta média como limiar para cada pixel dentro do bloco, e binariza-se o bloco.

2.2 EAN-13

O código EAN-13 é composto por 13 dígitos, sendo que na imagem em si, são codificados apenas 12 dígitos. Em relação ao significado da informação, o código pode ser dividido em 4 partes: prefixo GS1 de 3 dígitos (identifica um país), código do fabricante de tamanho variável, código do produto de tamanho variável, e o dígito de checagem. Do ponto de vista da imagem em si, o código é dividido em 5 partes: padrão de início, 6 dígitos do grupo esquerdo, padrão do meio, 6 dígitos do grupo direito, e padrão de término. Há também o dígito de checagem que pode ser obtido das combinações das paridades dos dígitos do grupo esquerdo.

Figura 4: Partes do código EAN-13: o dígito de checagem é 8; o grupo esquerdo é composto pelos dígitos 413000; o grupo direito é composto pelos dígitos 065504; os padrões de início, do meio e do fim podem ser percebidos pelas barras verticais maiores.

Considerando 1 uma barra preta, e 0 uma barra branca, o padrão de início e o de término são compostos por 101, enquanto que o padrão do meio é composto por 01010. Estes detalhes podem ser percebidos na Figura 4. O grupo da esquerda é codificado usando um padrão em que cada dígito tem duas possibilidades de codificação, um com paridade par (G), e outro com paridade ímpar (L). Cada dígito é composto por 7 barras verticais, então esta paridade considera a quantidade de barras pretas entre estas 7. O dígito de checagem é codificado indiretamente, selecionando um padrão de escolha das paridades entre os primeiros 6 dígitos. Por exemplo, na Figura 4, as paridades são LGLGGL, que representam o dígito 8. Já no grupo da direita, todos os dígitos são codificados considerando o mesmo padrão RRRRRR. Esses padrões de paridade e as codificações podem ser visulizados nas Tabelas 1 e 2. Note que uma entrada em R é o complemento bit a bit da entrada em L, e as entradas em G são as entradas em R na ordem reversa dos bits.

2.2.1 Algoritmo de Detecção e Decodificação do Código EAN-13

O algoritmo se inicia com a transformação da imagem para tons de cinza. Em seguida, ou a imagem é binarizada por completo ou a binarização é feita antes de cada linha ser processada. A segunda estratégia pode ser mais eficiente pois não necessariamente binariza toda a imagem para decodificar o código. Então, partindo da linha central, processa-se cada uma em busca das informações do código.

Primeiro Dígito	Grupo da Esquerda	Grupo da Direita
0	LLLLLL	RRRRRR
1	LLGLGG	RRRRRR
2	LLGGLG	RRRRRR
3	LLGGGL	RRRRRR
4	LGLLGG	RRRRRR
5	LGGLLG	RRRRRR
6	LGGGLL	RRRRRR
7	LGLGLG	RRRRRR
8	LGLGGL	RRRRRR
9	LGGLGL	RRRRRR

Tabela 1: Codificação do dígito de checagem dependendo das paridades dos dígitos do grupo da esquerda.

Dígito	Código L	Código G	Código R
0	0001101	0100111	1110010
1	0011001	0110011	1100110
2	0010011	0011011	1101100
3	0111101	0100001	1000010
4	0100011	0011101	1011100
5	0110001	0111001	1001110
6	0101111	0000101	1010000
7	0111011	0010001	1000100
8	0110111	0001001	1001000
9	0001011	0010111	1110100

Tabela 2: Códigos L, G e R para todos os dígitos.

Primeiramente, é procurado o padrão de ínicio 101 armazenando um contador de 3 posições para se contar a quantidade de pixels pretos e brancos em sequência. Para ser considerado o padrão de início, é preciso existir uma zona de segurança apenas com pixels brancos antes do início do padrão 101 com pelo menos o tamanho do padrão, para não ser um falso positivo. Também, é utilizada uma variância individual máxima de 0.7 entre os tamanhos das sequências de pixels. Isso porque, neste caso, cada barra no padrão 101 tem apenas uma unidade de largura, mas em padrões mais complexos poderia se ter 1100110, por exemplo. São somadas as variâncias de cada valor do contador em relação à respectiva quantidade de barras verticais no padrão, neste caso havendo apenas uma barra vertical para cada valor. Depois esta variância é dividida pela soma total dos contadores, e este valor também precisa ser menor do que outro limiar de 0.48. Este processo diminui a possibilidade de se existirem falsos positivos. Se por acaso algum critério não é satisfeito, então a procura pelo padrão de início continua.

Com o padrão de início encontrado, os 6 dígitos do lado esquerdo são procurados utilizando os valores da Tabela 2. Aqui também são utilizados contadores para se contar pixels pretos e brancos, e as variâncias citadas acima. Para cada padrão L ou G, é realizada uma tentativa de contagem dos pixels, e o padrão que tiver menor variância é considerado o melhor para o dígito. Durante o processo também são calculadas as paridades para se descobrir o dígito de checagem buscando-se em uma tabela.

Em seguida, é procurado o padrão do meio 01010, também utilizando contadores e as variâncias já citadas. Então, utilizando as tabelas são calculados os dígitos do lado direito seguindo um processo muito similar aos dígitos do lado esquerdo. Depois, é procurado o padrão final 101, também com uma região de segurança apenas com pixels brancos após o código. Assim, todos os 13 dígitos do código são decodificados.

Por fim, há apenas uma etapa final para se fazer um checksum dos dígitos, para saber se não houveram erros no processo de decodificação. Para este algoritmo, ignorando o dígito de checagem, multiplicam-se os dígitos por 1 e por 3, em uma sequência que se repete, e depois essa soma inicial é dividida por 10. O resultado é transformado em inteiro com a função piso, soma-se 1, multiplica-se o resultado por 10, se subtrai o valor da soma inicial das multiplicações e tira a função módulo 10. Se este resultado for igual ao dígito de checagem, então a decodificação não teve erros.

2.3 QR Code

O QR Code é uma matriz quadrada e foi desenvolvido para ser de rápida leitura. As informações estão dispostas em uma grade com fundo branco, e possui a característica adicional de utilização do código corretor de erros Reed-Solomon [14], utilizado para se corrigir possíveis erros na decodificação dos bytes na imagem. Os dados em si são extraídos de padrões presentes tanto na horizontal quanto na vertical. O código possui várias versões, de 1 a 40, que diferem no tamanho da matriz e da quantidade máxima de informações armazenadas, como pode ser visto na Figura 5.

(a) Versão 4 (33 x 33) (b) Versão 10 (57 x 57)

Figura 5: Diferentes versões do QR Code.

2.3.1 Código Reed-Solomon

Sobre o código corretor de erros, as palavras no código, denominadas de *codewords*, utilizam 8 bits com 4 níveis de correção. Quanto maior o nível de correção, menos informação o código é capaz de armazenar. Em seguida, estão os 4 níveis de correção:

- Nível L (Low): 7% das *codewords* podem ser restauradas.
- Nível M (Medium): 15% das *codewords* podem ser restauradas.
- Nível Q (Quartile): 25% das *codewords* podem ser restauradas.
- Nível H (High): 30% das *codewords* podem ser restauradas.

Para códigos maiores, a mensagem é quebrada em blocos de código Reed-Solomon. Os blocos depois são intercalados fora da ordem normal para que danos locais não comprometam a leitura. Com isso, é possível a adição de figuras dentro da própria matriz do código, sem que haja perda das informações, como a Figura 6. Detalhes sobre os algoritmos utilizados pelo código de Reed-Solomon, como divisão de polinômios e campo de Galois podem ser encontrados em [14].

Figura 6: QR Code com imagem inserida dentro do código.

2.3.2 Estrutura do QR Code

O QR Code e as informações visuais estão separadas em 7 partes: informações da versão, informações do formato, dados com chaves de correção de erro, 3 padrões de posição, número variável de padrões de alinhamento, padrões de timing, e quiet zone. Estas partes podem ser visualizadas na Figura 7. Também, na região inferior vertical do formato, sempre há um quadrado preto nas coordenadas (8, 4 * versão + 9) na grade do código, que é o quadrado mais alto nesta região.

- Versão: região que guarda a informação de qual versão é o QR Code da imagem, de 1 a 40.
- Formato: região que guarda qual o nível de correção de erros e qual o padrão de máscara a ser utilizada, que será explicado posteriormente.

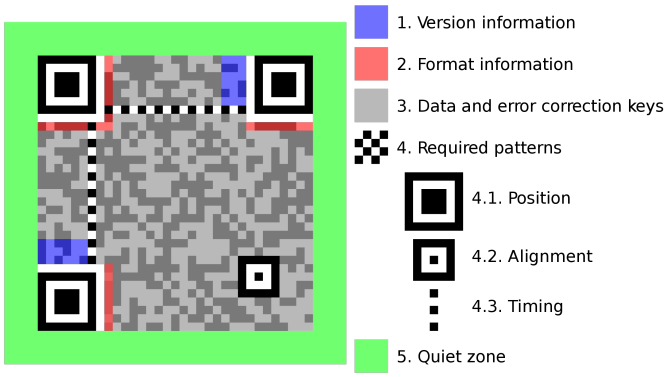


Figura 7: QR Code e as diversas informações na imagem. Imagem de Zephyris, distribuída sob a licença *Creative Commons Attribution-Share Alike 3.0 Unported*.

- Dados: armazena a informação do código propriamente dita separada em blocos, junto com chaves de correção de erros.
- Padrões de posição: 3 padrões utilizados para se encontrar o QR Code na imagem.
- Padrões de alinhamento: padrões utilizados para se alinhar o código, que juntamente com os padrões de posição, ajudam a corrigir problemas de distorção.
- Padrões de timing: uma linha horizontal e uma vertical que ajudam a detectar a posição de cada célula de dados, também chamadas de módulos, no QR Code.
- Quiet zone: zona branca de segurança ao redor do código.

As áreas destinadas a versão são compostas por 18 bits, dispostas em um retângulo 6×3. Códigos versão 7 ou maiores precisam necessariamente possuir os dois retângulos indicando as versões, mas versões menores não possuem os retângulos. As Tabelas 3, 4 e 5 mostram as ordens dos bits no retângulo 6×3, junto com os bits para versões 7 ou maiores. Estes bits utilizam o código Golay, que consiste em 6 bits para decodificar a versão e 12 bits de erro. A versão do código também define a quantidade de padrões de alinhamento, e as disposições dos blocos do código corretor de erro, mas como estas tabelas são grandes, elas podem ser visualizadas no código da biblioteca ZXing [1].

0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17

Tabela 3: Disposição dos bits de versão no retângulo inferior.

As áreas destinadas ao formato possuem 15 bits de informação, sendo que 2 bits são destinados ao nível de correção de erros, 3 bits para o padrão de máscara, e 10 bits para correção de erros. Na linha vertical inferior, estão os bits de 0 a 6 junto com o quadrado

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14
15	16	17

Tabela 4: Disposição dos bits de versão no retângulo superior.

preto citado anteriormente, e na área superior direita estão os bits 7 a 14. Na área superior esquerda estão os mesmos bits, começando do lado esquerdo e terminando no topo da imagem. Neste último caso são ignorados dois bits pretos do padrão de *timing*. Na Tabela 6 estão os bits do código corretor de erro, na Tabela 7 estão as máscaras, e na Tabela 8 estão as 32 possíveis combinações de bits para o formato. Estas máscaras são utilizadas para saber se a cor do bit atual deve ser trocada. Se o resultado da máscara for 0, então o bit é trocado, do contrário ele permanece o mesmo.

2.3.3 Algoritmo de Detecção e Decodificação do QR Code

O algoritmo se inicia com a transformação da imagem para tons de cinza. Em seguida, é utilizado um algoritmo de limiarização para binarizar a imagem, para se começar a etapa de detecção. Esta etapa se inicia com a busca dos padrões de posição, e para tornar a busca mais rápida no início, são puladas algumas linhas, caminhando de 3 em 3. Quando um padrão é encontrado, estes pulos são alterados para 2 em 2.

Estes padrões de posição obedecem uma razão 1:1:3:1:1, então são utilizados contadores para contar pixels pretos e brancos, igual o caso do código EAN-13. Quando uma sequência de preto/branco/preto/branco/preto é encontrada, ela é testada se calculando uma variância máxima segundo o seguinte algoritmo:

1. Soma os contadores e divide por 7.

2. Considera a variância máxima a metade do valor anterior, ou seja, permite menos de 50% de variância.

3. Testa cada contador, obedecendo as proporções 1:1:3:1:1.

Quando um possível padrão horizontal é encontrado, então é calculado o possível centro e é realizada uma contagem vertical checando se ela também obedece a proporção 1:1:3:1:1. Se na checagem vertical o tamanho é mais de 40% diferente do tamanho horizontal, assume que é um falso positivo. Então, é calculado o centro tanto na horizontal quanto na vertical.

Em seguida, com o centro encontrado, é feita novamente uma checagem horizontal. Isso é feito para se localizar o verdadeiro centro horizontal em casos de inclinações mais extremas. Aqui também é utilizado a checagem de tamanho 40% diferente. Então, é feita uma checagem diagonal pelas proporções passando pelo centro encontrado, para de fato

Versão	Bits em Hexadecimal
7	0x07C94
8	0x085BC
9	0x09A99
10	0x0A4D3
11	0x0BBF6
12	0x0C762
13	0x0D847
14	0x0E60D
15	0x0F928
16	0x10B78
17	0x1145D
18	0x12A17
19	0x13532
20	0x149A6
21	0x15683
22	0x168C9
23	0x177EC
24	0x18EC4
25	0x191E1
26	0x1AFAB
27	0x1B08E
28	0x1CC1A
29	0x1D33F
30	0x1ED75
31	0x1F250
32	0x209D5
33	0x216F0
34	0x228BA
35	0x2379F
36	0x24B0B
37	0x2542E
38	0x26A64
39	0x27541
40	0x28C69

Tabela 5: *Lookup* dos bits de versão em hexadecimal.

validar o padrão de posição. Esta checagem diagonal utiliza 75% de variância máxima, e não 50% igual aos outros casos.

Para não se encontrar o mesmo padrão de posição mais de uma vez, as posições e dimensões são comparadas. Em caso de serem próximas, os centros e o tamanho são combinados em uma nova estimativa. Também, para cada padrão ser considerado como válido,

Nível de Correção de Erro	Bits	Inteiro Equivalente
L	01	1
M	00	0
Q	11	3
H	10	2

Tabela 6: *Lookup* dos bits do nível de correção de erro e o inteiro equivalente.

Número da Máscara	Função da Máscara
0	(row + column) & 0x01 == 0
1	(row & 0x01) == 0
2	column % 3 == 0
3	(row + column) % 3 == 0
4	((floor(row / 2) + floor(column / 3)) & 0x01) == 0
5	(row * column) % 6 == 0
6	((row * column) % 6) <3
7	((row + column + ((row * column) % 3)) & 0x01) == 0

Tabela 7: Números e funções das máscaras.

ele tem que ser encontrado pelo algoritmo pelo menos duas vezes.

Em seguida, após encontrar três padrões, é checado se o tamanho destes padrões é similar. Quando o desvio total da média excede 5% do tamanho total, um dos padrões é considerado um falso positivo, e a busca continua. Caso vários padrões tenham sido encontrados, são escolhidos aqueles em que seus tamanhos variam menos em relação à média.

Com apenas 3 padrões de posição selecionados, eles são posicionados corretamente, ou seja, é definido qual deles é o superior direito (C), o superior esquerdo (B) e o inferior (A). Os pontos [A, B, C] são ordenados tal que AB é menor que AC e BC é menor que AC, e o ângulo entre BC e BA é menor que 180 graus.

Na sequência, é calculado o tamanho de cada quadrado na grade do código, também chamado de módulo. Para este cálculo, é feita uma média das estimativas do cálculo considerando duas duplas de pontos: superior esquerdo e superior direito, ou superior esquerdo e inferior. A ideia desta parte é traçar uma linha entre os centros, e calcular o tamanho de uma sequência de módulos preto/branco/preto. Para isso, é utilizada uma adaptação do algoritmo de Bresenham, que desenha linhas com ângulo de inclinação em grades. Considerando uma dupla de pontos, são feitas estimativas entre os pontos 1 e 2, mas também entre os pontos 2 e 1, também tirando uma média depois. O valor encontrado do tamanho do módulo é um ponto flutuante.

Com a estimativa do tamanho do quadrado, são calculadas as dimensões do código, também utilizando as duas duplas de pontos para se tirar uma média das distâncias entre os pontos, e adicionando 7. Então, é estimada a versão do código somente através de suas

Nível de Correção de Erro	Máscara	Bits em Hexadecimal
L	0	0x77C4
L	1	0x72F3
L	2	0x7DAA
L	3	0x789D
L	4	0x662F
L	5	0x6318
L	6	0x6C41
L	7	0x6976
M	0	0x5412
M	1	0x5125
M	2	0x5E7C
M	3	0x5B4B
M	4	0x45F9
M	5	0x40CE
M	6	0x4F97
M	7	0x4AA0
Q	0	0x355F
Q	1	0x3068
Q	2	0x3F31
Q	3	0x3A06
Q	4	0x24B4
Q	5	0x2183
Q	6	0x2EDA
Q	7	0x2BED
H	0	0x1689
H	1	0x13BE
H	2	0x1CE7
H	3	0x19D0
H	4	0x0762
H	5	0x0255
H	6	0x0D0C
H	7	0x083B

Tabela 8: *Lookup* para os bits em hexadecimal do formato, com os valores do nível de correção de erro e da máscara.

dimensões, isso para saber se aquele código tem um padrão de alinhamento. Caso o código possua o padrão, é feita uma estimativa do local do padrão de alinhamento mais a direita e mais abaixo na imagem, para não se precisar procurar na imagem toda. Algumas tentativas são feitas aumentando-se o raio de busca.

A busca pelo padrão de alinhamento é muito similar a do padrão de posição, com a diferença que as proporções buscadas são 1:1:1, diferentemente da 1:1:3:1:1 anterior. Quando

é encontrado um padrão, também é utilizada variância máxima permitida de 50%. Neste caso, também é necessário encontrar o padrão duas vezes, mas como checagem apenas é feita uma vertical, sem checar a diagonal e a horizontal novamente.

Em seguida, é feita uma transformação (*perspective transform*) na imagem para corrigir distorções e rotações na imagem. Para a transformação, são utilizados os padrões de posição e alinhamento para mapear os cálculos, mas se o código não possui o padrão de alinhamento, é utilizado o ponto mais a direita e mais abaixo do código para se ter quatro pontos. A implementação desta transformada foi baseada em [15]. Enquanto os pontos são transformados, há uma checagem para ver se eles são transformados para dentro da imagem, e alguns casos são tratados. Com isso, os bits do código são extraídos e a etapa de detecção do algoritmo termina.

Para a etapa de decodificação, primeiramente são decodificadas as informações da versão do QR Code. Se as dimensões forem menores do que as da versão 7 (45×45), então a versão já é retornada. Do contrário, procura-se nos locais que mantém a informação da versão. Primeiramente procura-se no canto superior direito, e caso falhe a decodificação, é procurado no campo inferior esquerdo. Para se definir o número da versão, é utilizada a tabela de *lookup* 5. Se os bits não forem exatamente iguais, é utilizada a versão com menor número de bits diferentes, com um máximo de 3 bits errados permitidos.

Em seguida, são lidas as informações do formato. Os bits das duas regiões com informações de formato são colhidos e o procedimento é similar ao caso da versão. É utilizada a Tabela 8, e o valor que possuir menos bits diferentes é utilizado, também com um máximo de 3 bits errados permitidos. Assim, são extraídas as informações do nível de correção de erros e da máscara.

Na sequência, são lidas as *codewords*. Cada versão de código tem formatos diferentes de *codewords*, devido a quantidades diferentes de padrões de alinhamento e a dimensão do código ser diferente. Primeiramente, é aplicada a máscara para se recuperar os bits originais. Neste momento são calculadas as posições a serem ignoradas durante a leitura dos dados, que são os padrões de posição, os padrões de alinhamento, os padrões de timing, as informações de formato, e para códigos versão 7 ou superior as informações da versão. Então, é feita a leitura das *codewords* propriamente ditas. A leitura se inicia no canto inferior direito do código, e segue padrões como por exemplo a Figura 8. Note que, neste exemplo, as *codewords* estão numeradas, e não seguem a ordem natural da informação.

Em seguida, a ordem das *codewords* é corrigida e elas são separadas em blocos de dados (*datablocks*). A partir da informação de versão, é possível saber quantos blocos de *codewords* existem e quantas *codewords* de dados e de erro existem no máximo em cada *datablock*. Alguns *datablocks* podem não possuir a mesma quantidade de *codewords* que outros. Depois, com os blocos separados e a ordem corrigida, para cada datablock é aplicado o código corretor de erros Reed-Solomon. Então, as *codewords* que representam os dados são unidas em um stream de bytes, ignorando-se as *codewords* de correção de erro que já foram utilizadas.

Por fim, o *stream* de bytes é decodificado. Dentro do mesmo QR Code, é possível a utilização de diferentes modos de leitura. Então, os primeiros quatro bits são lidos para se saber qual o modo que o segmento seguinte está codificado. Os modos disponíveis são listados a seguir:

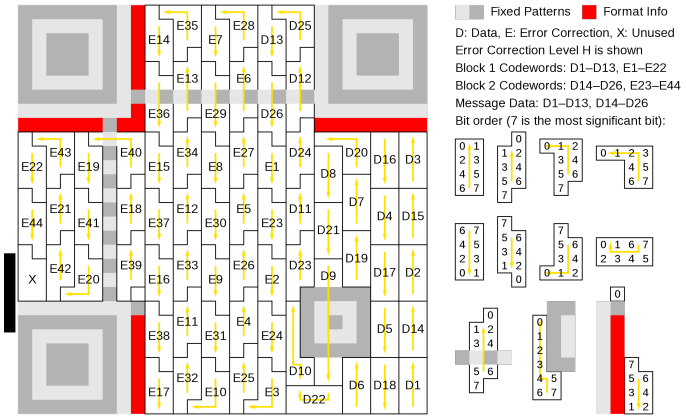


Figura 8: Exemplo de direções de leitura de *codewords* no QR Code, junto com os formatos e a disposição espacial da informação.

- 0x0 - TERMINATOR: Indica fim de leitura.
- 0x1 - NUMERIC: Números apenas.
- 0x2 - ALPHANUMERIC: Caracteres alfanuméricos.
- 0x3 - STRUCTURED_APPEND: Não suportado.
- 0x4 - BYTE: Neste caso, informações são usadas para se assumir uma codificação. Esta codificação pode ser de vários tipos diferentes: UTF8, ASCII, SJIS, ISO8859_1, etc. Para mais detalhes sobre o algoritmo que assume a codificação veja [1].
- 0x5 - FNC1_FIRST_POSITION: Primeiro caractere especial que separa sequências de tamanho variável.
- 0x7 - ECI: Diz informações sobre qual a codificação dos próximos bytes. Esta codificação pode ser de vários tipos diferentes: UTF8, ASCII, SJIS, ISO8859_1, etc.
- 0x8 - FNC1_SECOND_POSITION: Segundo caractere especial que separa sequências de tamanho variável.
- 0x9 - KANJI: Caracteres de língua japonesa.
- 0xD - HANZI: Caracteres chineses.

Se os passos anteriores não apresentarem falhas, então as informações do código foram decodificadas corretamente. Porém, se alguma parte da etapa de decodificação falhou, existe ainda a possibilidade do código estar espelhado. Então, é aplicada uma transposição da matriz do código e é tentada a decodificação novamente. Assim, termina-se o algoritmo do QR Code.

3 Métodos Propostos

A partir dos algoritmos de detecção e decodificação do código EAN-13 e do QR Code, foi proposta uma mudança no início do algoritmo, antes de aplicar o algoritmo de limiarização. A ideia foi realizar um pré processamento com filtros realçadores de borda após a alteração de cor da imagem para tons de cinza. A função esperada destes filtros é a diminuição do efeito de possível borramento nas imagens, que prejudica os algoritmos de limiarização. Foram utilizados dois filtros nos experimentos, um 3×3 e um 5×5, que são mostrados a seguir:

$$kernel_{3\times3} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

$$kernel_{5\times5} = \frac{1}{8}\begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & 2 & 8 & 2 & -1 \\ -1 & 2 & 2 & 2 & -1 \\ -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

4 Resultados Experimentais

Nesta seção, são mostrados os resultados dos experimentos na leitura do código EAN-13 e QR Code. Para o caso do QR Code, foram criadas 6 imagens aplicando borramento Gaussiano, para saber se os filtros realçadores de borda ajudariam na detecção e decodificação destes casos. Foram utilizadas 157 imagens para o código EAN-13 e 184 imagens para o QR Code. Como foram utilizadas as imagens disponíveis na biblioteca ZXing [1], foi mantida a separação de imagens em 5 grupos para o EAN-13 e 7 grupos (6 da biblioteca e um novo grupo com borramento Gaussiano) para o QR Code. Os resultados com as acurácias de acerto na leitura estão nas Tabelas 9 e 10.

A partir dos experimentos, pode-se perceber que para o código EAN-13, as duas melhores abordagens foram o limiar total do ZXing com kernel 3×3 (0.78) e a implementação padrão da biblioteca, o limiar da linha do ZXing (0.75). Outra abordagem que teve um bom desempenho foi o limiar adaptativo da média com kernel 3×3 (0.73). Analisando grupo a grupo, pode-se perceber que o método da biblioteca possuía scores mais altos em algumas categorias, mas caía muito a acurácia em outras. A abordagem do limiar total com kernel 3×3 foi bem mais constante, e obteve resultados altos em mais grupos.

Com isso, foi possível concluir que houve um ganho ao utilizar o pré processamento com filtro de realce de bordas, mas esse aumento depende muito também do método de limiarização escolhido, pois o limiar da linha do ZXing com os kernels tiveram resultados muito inferiores quando comparados com a implementação sem o kernel (0.28 e 0.47). Também é importante notar o grupo 5 que possuía imagens com muito borramento. Como todos os métodos tiveram acurácia 0.00, o borramento ainda é um problema para os algoritmos do EAN-13.

Método de Limiarização	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Total
ZXing Linha	0.94	0.61	1.00	0.64	0.00	0.75
ZXing Total	0.85	0.57	0.95	0.59	0.00	0.70
Simples	0.71	0.32	0.85	0.18	0.00	0.54
Otsu	0.79	0.43	0.95	0.18	0.00	0.61
Adaptativa Média	0.91	0.57	0.96	0.23	0.00	0.67
Adaptativa Gaussiana	0.88	0.68	0.93	0.50	0.00	0.71
ZXing Linha + Kernel 3×3	0.24	0.68	0.13	0.45	0.00	0.28
ZXing Total + Kernel 3×3	0.85	0.82	0.98	0.77	0.00	0.78
Simples + Kernel 3×3	0.68	0.39	0.98	0.23	0.00	0.59
Otsu + Kernel 3×3	0.79	0.61	1.00	0.50	0.00	0.70
Adaptativa Média + Kernel 3×3	0.62	0.75	0.98	0.82	0.00	0.73
Adaptativa Gaussiana + Kernel 3×3	0.38	0.61	0.96	0.55	0.00	0.61
ZXing Linha + Kernel 5×5	0.68	0.71	0.40	0.41	0.00	0.47
ZXing Total + Kernel 5×5	0.88	0.82	0.38	0.59	0.00	0.55
Simples + Kernel 5×5	0.76	0.36	0.55	0.18	0.00	0.45
Otsu + Kernel 5×5	0.91	0.61	0.65	0.41	0.00	0.59
Adaptativa Média + Kernel 5×5	0.76	0.79	0.35	0.59	0.00	0.51
Adaptativa Gaussiana + Kernel 5×5	0.53	0.75	0.25	0.27	0.00	0.38

Tabela 9: Acurácia para os diferentes experimentos com o código EAN-13.

Método de Limiarização	Grupo 1	Grupo 2	Grupo 3	Grupo 4	Grupo 5	Grupo 6	Grupo Novo	Total
ZXing Total	0.85	0.88	0.90	0.75	1.00	1.00	0.00	0.84
Simples	0.85	0.76	0.79	0.40	0.37	1.00	0.50	0.65
Otsu	0.85	0.82	0.83	0.50	0.58	1.00	0.33	0.72
Adaptativa Média	0.85	0.74	0.88	0.77	1.00	0.93	0.00	0.81
Adaptativa Gaussiana	0.95	0.82	0.88	0.75	1.00	0.93	0.17	0.84
ZXing Total + Kernel 3×3	1.00	0.71	0.93	0.67	1.00	0.93	0.83	0.83
Simples + Kernel 3×3	0.90	0.68	0.69	0.40	0.26	1.00	0.83	0.62
Otsu + Kernel 3×3	0.65	0.71	0.90	0.56	0.58	1.00	1.00	0.73
Adaptativa Média + Kernel 3×3	0.55	0.65	0.86	0.67	1.00	0.93	0.33	0.74
Adaptativa Gaussiana + Kernel 3×3	1.00	0.68	0.88	0.63	1.00	0.93	0.50	0.79
ZXing Total + Kernel 5×5	1.00	0.88	0.90	0.73	0.79	1.00	0.83	0.86
Simples + Kernel 5×5	0.90	0.76	0.81	0.52	0.37	1.00	0.83	0.71
Otsu + Kernel 5×5	0.90	0.79	0.90	0.73	0.47	1.00	1.00	0.80
Adaptativa Média + Kernel 5×5	0.95	0.79	0.93	0.79	0.79	0.93	0.33	0.84
Adaptativa Gaussiana + Kernel 5×5	1.00	0.82	0.93	0.73	0.79	0.93	0.50	0.84

Tabela 10: Acurácia para os diferentes experimentos com o QR Code.

Em relação ao QR Code, pode-se perceber que muitos métodos tiveram acurácia entre 0.83 e 0.86. Em valor absoluto, o melhor método foi o limiar total do ZXing com kernel 5×5, seguido pelo limiar total sem kernel. Com isso pode-se perceber que houve um pequeno ganho adicionando o realce de bordas nesse caso, porém, desconsiderando as imagens criadas, a acurácia sem kernel seria melhor.

Entretanto, analisando grupo a grupo, o limiar total com kernel 5×5 obteve resultados

melhores que o método da biblioteca no grupo 1, e foi um pouco pior no grupo 5. As imagens do grupo 5 possuem iluminação que varia, e também algumas imagens muito pequenas (177×177) de códigos de versão grande, como a Figura 9. Se estes casos não muito comuns forem ignorados, o método com kernel 5×5 se sairia melhor do que sem o kernel. Assim, é possível concluir que o pré processamento no QR Code também resulta em um pequeno ganho, mas esse sendo em casos mais específicos.

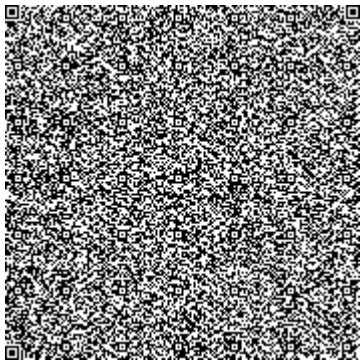


Figura 9: Imagem com QR Code de versão grande, mas com poucos pixels (177×177).

5 Conclusões e Trabalhos Futuros

Após a tradução do código da biblioteca do ZXing e os experimentos realizados, pode-se concluir que os métodos de pré processamento de realce de bordas aplicados antes dos algoritmos de limiarização nas imagens dos códigos EAN-13 e QR Code ajudaram a melhorar a acurácia nos resultados obtidos. As melhorias no código EAN-13 foram um pouco maiores, sendo que no QR Code foi necessário considerar alguns casos extras de borramento e ignorar outros casos menos comuns, como imagens muito pequenas.

O estudo deste tema permitiu verificar a complexidade e a quantidade de algoritmos diferentes integrados para o processamento dos códigos, especialmente no caso do QR Code. Como trabalhos futuros, o código EAN-13 ainda é muito sensível a borramentos e mais modificações poderiam ser feitas para se melhorar esta etapa. A pesquisa realizada também permitiu encontrar métodos utilizando redes neurais para o processamento dos códigos em [5] e [12], que poderiam ser um ponto inicial para futuros refinamentos nos métodos.

Referências

[1] ZXing ("Zebra Crossing") Barcode Scanning Library for Java, Android. `https://zxing.org/w/decode.jspx`.

[2] Ciażyński, Karol & Fabijańska, Anna. (2015). Detection of QR-Codes in Digital Images Based on Histogram Similarity. Image Processing & Communications, 20(2), pp. 41–48.

[3] P. Liyanage, J. (2007). Efficient Decoding of Blurred, Pitched, and Scratched Barcode Images.

[4] Xu, Mingliang & Li, Qingfeng & Niu, Jianwei & Liu, Xiting & Xu, Weiwei & Lv, Pei & Zhou, Bing. (2018). ART-UP: A Novel Method for Generating Scanning-robust Aesthetic QR codes. arXiv:1803.02280.

[5] Zamberletti, Alessandro & Gallo, Ignazio & Albertini, Simone. (2013). Robust Angle Invariant 1D Barcode Detection. IAPR Asian Conference on Pattern Recognition, pp. 160–164.

[6] Chen, Weibing & Yang, Gaobo & Zhang, Ganglin. (2012). A Simple and Efficient Image Pre-processing for QR Decoder. 2nd International Conference on Electronic & Mechanical Engineering and Information Technology, pp. 234–238.

[7] Chen, Qichao & Du, Yaowei & Lin, Risan & Tian, Yumin. (2012). Fast QR Code Image Process and Detection. 305-312. 10.1007/978-3-642-32427-7_42.

[8] Creusot, Clement & Munawar, Asim. (2015). Real-Time Barcode Detection in the Wild. IEEE Winter Conference on Applications of Computer Vision. pp. 239–245.

[9] Szentandrási, István & Herout, Adam & Dubska, Marketa. (2013). Fast Detection and Recognition of QR Codes in High-Resolution Images. 28th Spring Conference on Computer Graphics, pp. 129–136.

[10] Tong, Lingling & Gu, Xiaoguang & Dai, Feng. (2014). QR Code Detection Based on Local Features. International Conference on Internet Multimedia Computing and Service, p. 319.

[11] Tribak, Hicham & Zaz, Youssef. (2017). QR Code Recognition based on Principal Components Analysis Method. International Journal of Advanced Computer Science and Applications, 8(4).

[12] Kold Hansen, Daniel & Nasrollahi, Kamal & B. Rasmusen, Christoffer & Moeslund, Thomas. (2017). Real-Time Barcode Detection and Classification using Deep Learning. International Joint Conference on Computational Intelligence. pp. 321–327.

[13] Otsu, Nobuyuki. (1979). A Threshold Selection Method from Gray-Level Histograms. IEEE Transactions on Systems, Man and Cybernetics, 9(1), pp. 62–66.

[14] Reed, Irving S. & Solomon, Gustave (1960). Polynomial Codes over Certain Finite Fields. Journal of the Society for Industrial and Applied Mathematics (SIAM), 8(2), pp. 300–304.

[15] Wolberg, George. (1990). Digital Image Warping. IEEE Computer Society Press.