INTRODUCTION

From reference [1] we present Figs 1 thru 3:

**QR code**

From Wikipedia, the free encyclopedia

A QR code (abbreviated from Quick Response code) is a type of matrix barcode (or two-dimensional code) first designed for the automotive industry. More recently, the system has become popular outside of the industry due to its fast readability and large storage capacity compared to traditional UPC barcodes. The code consists of black modules arranged in a square pattern on a white background. The information encoded can be made up of four standardized kinds (“modes”) of data (numeric, alphanumeric, byte-binary, Kanji), or by supported extensions virtually any kind of data.[1]

Created by Toyota subsidiary Denso Wave in 1994[2] to track vehicles during the manufacturing process, the QR code is one of the most popular types of two-dimensional barcodes. It was designed to allow its contents to be decoded at high speed.[3]

Figure 1: Introduction from wikipedia on QR code.

The layout of the QR code includes three position markers (upper left, lower left and upper right corners), one alignment marker (lower right) and two timing sequences (vertical and horizontal) as shown in Fig. 2.

Figure 2: Position, alignment and timing features.

The QR code is an ISO standard. The information, depending on the version, may be in 3x2 blocks as shown in Fig. 3.
Message placement within a QR symbol

Figure 3: Data layout of a QR code.

For this visualization we will use the position, alignment and timing features of the QR code but replace the information code with a random encryption. This will demonstrate the concepts without the complexity of programming the ISO standard. The encryption technique will be a standard exclusive XOR method with a binary image and binary key. The ideal QR code with our coded pattern will be upscaled to simulate a camera with about 10 pixels across 1 QR pixel spacing. The camera resolution image will be blurred with a deterministic Gaussian function to simulate an ideal lens spread function. An additive white Gaussian noise (AWGN) will be implemented to simulate the camera pixel noise.

BEGINNING OF TASK

Please follow the tutorial and reproduce the figures with your own code. Use your own input image for your work.

The first step is to setup the size of the QR code and form a basic mask. We will also input an image to be encrypted. The student is asked to provide their own image for encryption.

% QR code Visualization
clear all;
% define mask with square rings
Nx=50; % number of pixels along horizontal and vertical of QR code
Ncamera=10; % number of camera pixels per QR code pixel
My=Nx;
Amask=zeros(My,Nx);
%input binary image
A_bmp=double(imread('SampleImage.bmp')); % load example.bmp image
Ar=A_bmp(:,:,1);
Ag=A_bmp(:,:,2);
Ab=A_bmp(:,:,3);
Aimage=binarize((Ar+Ag+Ab)/3);
[My1 Nx1]=size(Aimage);

The next task is to establish the positioning features using square “rings” and an existing .m function called DrawRings.

% define feature shapes 1, 2 and 3 to have Nrings
Nrings=4;
DA(1)=0;DB(1)=3;CA(1)=1;CB(1)=0; % Inside ring is a black box
DA(2)=3;DB(2)=5;CA(2)=0;CB(2)=1; % Thin ring is white
DA(3)=5;DB(3)=7;CA(3)=1;CB(3)=0;
DA(4)=7;DB(4)=9;CA(4)=0;CB(4)=1;
% feature positions
NfeatureA=3;
xc(1)=4;yc(1)=4;
xc(2)=1+Nx-xc(1);yc(2)=yc(1);
xc(3)=xc(1);yc(3)=1+My-yc(1);

Each positioning feature has 4 “rings” which are defined above by letting DA be the “diameter” inside the ring and DB be the outer diameter that is the ring. Note that DA is usually DB-2 in order to create a “ring” that is one pixel wide. There are 3 position features and are drawn with

% draw features 1, 2 and 3
for f=1:NfeatureA;
xc2=xc(f);yc2=yc(f);
Amask= DrawRings(Amask,DA,DB,CA,CB,xc2,yc2,Nrings);
end;

There is one alignment feature which only has 3 rings and is superimposed onto the mask with

% alignment feature has 3 rings
Nrings=3;
DA(1)=0;DB(1)=1;CA(1)=1;CB(1)=0; % Inside ring is a black box
DA(2)=1;DB(2)=3;CA(2)=0;CB(2)=1; % Thin ring is white
DA(3)=3;DB(3)=5;CA(3)=1;CB(3)=0;
xc2=Nx-6;yc2=My-6;
Amask= DrawRings(Amask,DA,DB,CA,CB,xc2,yc2,Nrings);

Notice the position is referenced to Nx, My to allow size changes in the QR code map. The last features are horizontal and vertical timing features which are lines of alternating “1” and “0” values such that:

% Timing features
x1=7;y1=9;
x2=7;y2=My-8;
dash0=0;dash1=1;
Amask=DrawDashLine(Amask,x1,y1,x2,y2,dash1,dash0);
x1=9;y1=7;
x2=Nx-8;y2=7;
dash0=0;dash1=1;
Amask=DrawDashLine(Amask,x1,y1,x2,y2,dash1,dash0);
We are now ready to view Amask using

```matlab
figure(1)
imagesc(Amask)
colormap gray;
axis image;
title('QR chart feature map')
print -djpeg fig1
```

such that

![QR chart feature map](image)

**Figure 4: Feature positions of QR code.**

The encryption is based on a binary key in the same dimensions as the input image. The key is positioned onto the QR field and formed as:

```matlab
% ENCRYPTION PATTERN KEY
xulKey=9;
yulKey=9;
NxKey=(Nx-xulKey);
MyKey=(My-2*yulKey);
Key=rand(MyKey,NxKey);
Key=binarize(Key);
```

The encoding key is shown in Fig. 5.
The image to be encoded is first scaled to be the same dimensions as the key and is shown in Fig. 6.

```matlab
% sample input image to size of Key
ax=Nx1/NxKey;
ay=My1/MyKey;
Bimage=zeros(MyKey,NxKey);
for m=1:MyKey;
    for n=1:NxKey;
        m1=floor(m*ay);
        n1=floor(n*ax);
        Bimage(m,n)=Aimage(m1,n1);
    end;
end;
```

Notice that the image scaling is done by looping through the output. This is a common method of transforming images because it prevents “pin holes” from forming in the output which would happen if the indexing was performed on the input image and mapped to the output.

The encryption is performed by XOR operation between the input image and the key such that:

```matlab
% encode image
for m=1:MyKey;
    for n=1:NxKey;
        if Bimage(m,n)==Key(m,n)
            Cimage(m,n)=0;
        else
            Cimage(m,n)=1;
        end;
    end;
end;
```
The encrypted image is then mapped to the QR field region as not to interfere with the QR features.

% map encrypted pattern onto QR code
for m=1:MyKey;
    for n=1:NxKey;
        Amask(m+yulKey,n+xulKey)=Cimage(m,n);
    end;
end;

We want to treat the QR code as an image captured by a high resolution camera so we must upsample it to the camera resolution, blur the image and add sensor noise. Upsampling is conveniently achieved by a Kronecker multiplication in MATLAB implemented by the C=kron(A,B) function. The upsampled QR code is shown in Fig. 7. The matrix B is a Ncamera x Ncamera unity matrix such that:

% upsample to higher resolution model
NxCam=Nx*Ncamera;
MyCam=My*Ncamera;
kernel=ones(Ncamera,Ncamera);
Cmask=kron(Amask,kernel);

Figure 7: Upsampled QR code.

Try reversing the order of Amask and kernel in kron and see what happens? Do you understand why? If not, search kronecker multiplication.

If the image in Fig. 7 were photographed by a high resolution camera, where 1 pixel in Fig. 7 spans Ncamera pixels of the camera, the first corruption of the image field would be blurring by
the camera lens. A common model of lens blurring is the Gaussian function or Gaussian “spread function.” This spread function is generated as follows and shown in Fig. 8.

```matlab
% gaussian filter
sigma=4;% spread function sigma;
x=1:NxCam;
x=x-NxCam/2;
y=1:MyCam;
y=y-MyCam/2;
% blurring
hgauss=gaperture(sigma,sigma,x,y);
```

Figure 8: Gaussian spread function.

The blurring is implemented by convolution via the DFT domain. This means that the spread function origin must be shifted to the upper left corner before performing the circular convolution.

```matlab
hgauss=fftshift(hgauss); % shift to put origin at the corners
% filter the large image
Dmask=real(ifft2(fft2(Cmask).*fft2(hgauss)));
Dmask=Dmask-min(min(Dmask));
Dmask=Dmask./max(max(Dmask));
```

The resulting image from the lens is shown in Fig. 9.

Figure 9: Blurred image.
In addition to blurring, there may be sensor noise due to low light levels. A common model of sensor noise is the Additive White Gaussian Noise (AWGN) model. AWGN is implemented as

```matlab
% Camera sensor noise
sigma=0.1;  % gaussian noise sigma;
noiseF=randn(MyCam,NxCam);
noiseF=noiseF*sigma;
% add noise
Dmask=Dmask+noiseF;
```

The resulting image coming from the sensor array is shown in Fig. 10.

![Sensor Noise](image)

**Figure 10: Blurred and noisy image.**

We will assume we have software functions that will align and orientate the image in shown in Fig. 10. Given that information we crop out the encoded region such that:

```matlab
% crop the image from QR code
Emask=zeros((MyKey*Ncamera),(NxKey*Ncamera));
Fmask=Emask;
for m=1:(MyKey*Ncamera);
    for n=1:(NxKey*Ncamera);
        Emask(m,n)=Dmask(m+(yulKey*Ncamera),n+(xulKey*Ncamera));
    end;
end;
```

The image to be decoded is shown in Fig. 11.
Because we want to understand what is “behind” the processing, we will decode at the camera resolution rather than downsample Fig. 11. So we upsample the key such that:

% decode image
Emask=binarize(Emask);
KeyCam=kron(Key,kernel);

To decode an XOR encryption, the key is simply XORed again with the encoded patterns such that:

```matlab
for m=1:(MyKey*Ncamera);
    for n=1:(NxKey*Ncamera);
        if Emask(m,n) == KeyCam(m,n)
            Fmask(m,n)=0;
        else
            Fmask(m,n)=1;
        end;
    end;
end;
```
The result in Fig. 13 reveals the edge affects between spatial pixels.

![Decoded Image](image1)

**Figure 13:** Decoded image prior to sampling within the spatial pixel "eye" regions.

If we downsample from the center of each spatial pixel, the result closely matches the original input.

```
% downsample from bit centers
% Decoded Image
Gmask=zeros(MyKey,NxKey);
for m=1:MyKey;
    for n=1:NxKey;
        mc= floor((m-1)*Ncamera+Ncamera/2);
        nc= floor((n-1)*Ncamera+Ncamera/2);
        Gmask(m,n)=Fmask(mc,nc);
    end;
end;
```

![Downsampled Decoded Image](image2)

**Figure 14:** Decoded image.

The decoded image in Fig. 14 has some bit errors. An objective measure of bit error is the Hamming distance. In this case it was 14 bit differences and implemented as:
% Hamming distance
Hammingdist=sum(sum(abs(Bimage-Gmask)))

>14

Try changing the sigma values in blurring and the noise to reduce the Hamming distance. Try making it larger. Try setting one sigma to 0 and see what effect the other has on image corruption?

REFERENCES