第3章 数字图像加密系统

本章将介绍常规的基于混沌系统的图像加密系统，并介绍常用的一些置乱算法与扩散算法。不失一般性，本章中混沌图像密码系统使用了超混沌 Lorenz 系统（如式(2-12)所示）产生密码。

3.1 图像加密与解密方案

基于混沌系统的数字图像加密与解密方案如图 3-1 所示。

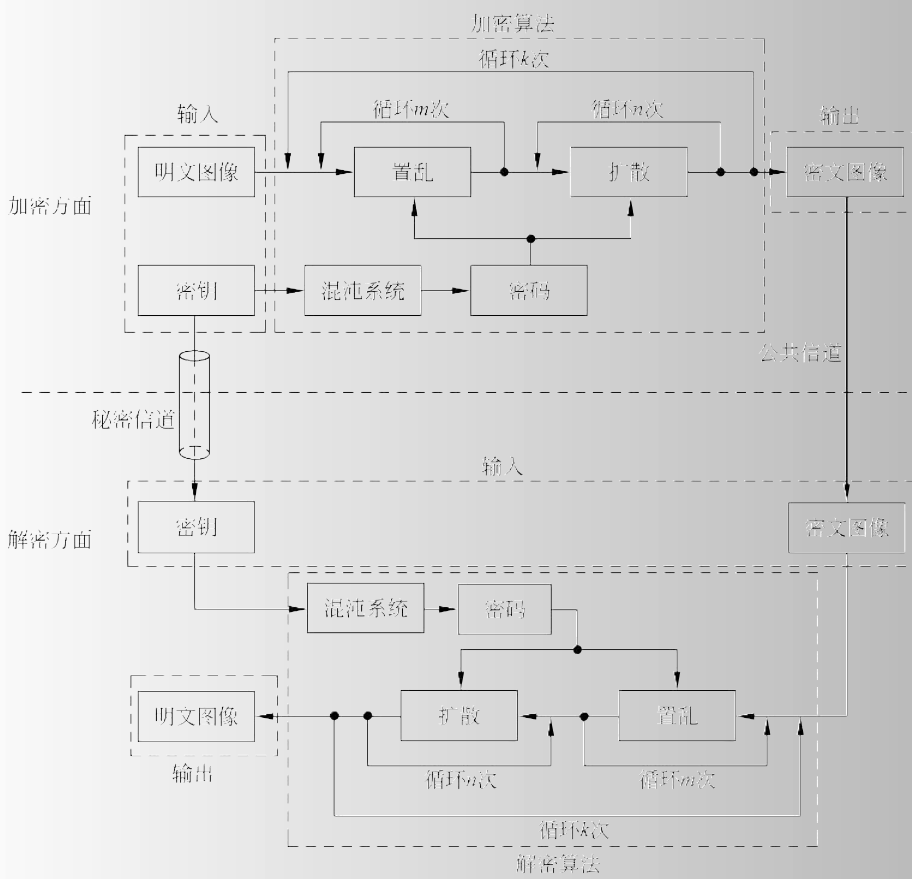


图 3-1 混沌数字图像密码系统

基于混沌系统的数字图像加密系统隶属于对称密钥密码系统，加密方面与解密方面具有相同的密钥。由图 3-1 可知，加密系统方面输入明文图像和密钥后，经过加密算法，输出密文图像；加密方面通过秘密信道将密钥传送给解密方面(大多数情况下，借助于公钥密码系统实现)，同时，通过公共信道将密文图像传送给解密方面。解密系统方面输入密钥和密文图像，经过解密算法，输出解密后的图像，只有当密钥正确时，才能输出还原后的明文图像。

一般地，解密算法是加密算法的逆过程。这里的“置乱”是指将图像的像素点位置变换，但每个像素点的值保持不变；“扩散”是指不改变像素点的位置，而是通过改变像素点的灰度值，使得任一像素点的灰度信息影响尽可能多的其他像素点的灰度值。很多情况下，置乱与扩散算法融合在一起，即同时改变像素点的位置和灰度值，使得任一像素点的灰度信息隐藏在尽可能多的其他像素点中。

图像的置乱与扩散可以在空间域中进行，也可以在频谱域中进行，而且在频谱域中可同时实现加密与压缩处理。本书中重点讨论空间域中的置乱与扩散算法，这些算法可以推广到频谱域中。

在本章中，明文图像均为 8b 的灰度图像，记为 $\boldsymbol{P}$，其大小记为 $M\times N$，表示 M 行 N 列的矩阵。超混沌 Lorenz 系统的参数和初始值用作密钥。

3.2 置乱算法

常用的置乱算法分为三类：其一，对二维图像矩阵进行行置乱和列置乱，或交叉进行行、列置乱；其二，将二维图像展开成一维行向量或一维列向量，对该向量进行位置置乱；其三，借助于 2×2 置乱矩阵变换二维图像的各个像素点的位置。本节将介绍这三类方法，并通过置乱后的图像直观地对比它们的处理效果。

1. 二维图像直接行置乱与列置乱

1) 随机置乱方法

行随机置乱：借助于超混沌 Lorenz 系统产生长度为 M 的随机数向量 $\boldsymbol{X}$，每个随机数 $X_i\in\{1,2,\cdots,M\}$。然后，图像矩阵 $\boldsymbol{P}$ 的第 i 行与第 X_i 行互换，$i=1,2,\cdots,M$。

列随机置乱：借助于超混沌 Lorenz 系统产生长度为 N 的随机数向量 $\boldsymbol{Y}$，每个随机数 $Y_j\in\{1,2,\cdots,N\}$。然后，图像矩阵 $\boldsymbol{P}$ 的第 j 列与第 Y_j 列互换，$j=1,2,\cdots,N$。

程序段 3-1　随机置乱方法实现行置乱与列置乱

```
clc;clear;
P = imread('4.2.04.tiff');P = rgb2gray(P);
iptsetpref('imshowborder','tight');
figure(1);imshow(P);
[M,N] = size(P);P = double(P);

n = M + N;
h = 0.002;t = 800;a = 10;b = 8/3;c = 28;r = -1;x0 = 1.1;y0 = 2.2;z0 = 3.3;w0 = 4.4;
s = zeros(1,n);
```

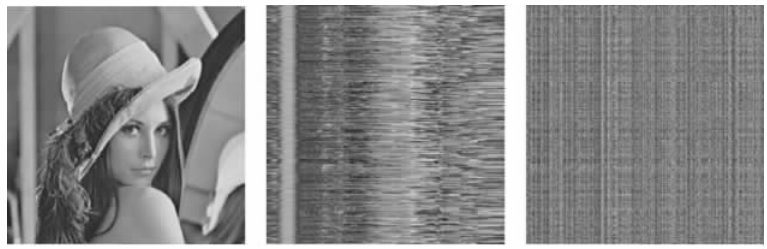
```
for i = 1:n + t
    K11 = a * (y0 - x0) + w0;K12 = a * (y0 - (x0 + K11 * h/2)) + w0;
    K13 = a * (y0 - (x0 + K12 * h/2)) + w0;K14 = a * (y0 - (x0 + K13 * h)) + w0;
    x1 = x0 + (K11 + K12 + K13 + K14) * h/6;
    K21 = c * x1 - y0 - x1 * z0;K22 = c * x1 - (y0 + K21 * h/2) - x1 * z0;
    K23 = c * x1 - (y0 + K22 * h/2) - x1 * z0;K24 = c * x1 - (y0 + K23 * h) - x1 * z0;
    y1 = y0 + (K21 + K22 + K23 + K24) * h/6;
    K31 = x1 * y1 - b * z0;K32 = x1 * y1 - b * (z0 + K31 * h/2);
    K33 = x1 * y1 - b * (z0 + K32 * h/2);K34 = x1 * y1 - b * (z0 + K33 * h);
    z1 = z0 + (K31 + K32 + K33 + K34) * h/6;
    K41 = - y1 * z1 + r * w0;K42 = - y1 * z1 + r * (w0 + K41 * h/2);
    K43 = - y1 * z1 + r * (w0 + K42 * h/2);K44 = - y1 * z1 + r * (w0 + K43 * h);
    w1 = w0 + (K41 + K42 + K43 + K44) * h/6;

    x0 = x1;y0 = y1;z0 = z1;w0 = w1;
    if i > t
        s(i - t) = x1;
        if mod((i - t),3000) == 0
          x0 = x0 + h * sin(y0);
    end
  end
end

X = mod(floor((s(1:M) + 100) * 10 ^10),M) + 1;
Y = mod(floor((s(M + 1:M + N) + 100) * 10 ^10),N) + 1;
A = P;
for i = 1:M
    t = A(i,:);A(i,:) = A(X(i),:);A(X(i),:) = t;
end
figure(2);imshow(uint8(A));
B = A;
for j = 1:N
    t = B(:,j);B(:,j) = B(:,Y(j));B(:,Y(j)) = t;
end
figure(3);imshow(uint8(B));
```

在程序段 3-1 中，第 2 行读入图像 4.2.04.tiff(512×512 像素点的 Lena 图像)，将其转化为灰度图像，保存在变量 ***P*** 中。第 3 行设置图像显示时无边框。第 4 行显示 Lena 图像。第 5 行获得 Lena 图像矩阵的行数 M 与列数 N。第 7 行设定计算的混沌序列的长度 n。第 8～31 行生成混沌序列，保存在变量 s 中，此时 s 中保存的伪随机数为浮点数。第 27～29 行在每 3000 次迭代后对混沌状态 x0 进行小的扰动。第 33～34 行生成行置乱向量 ***X*** 和列置乱向量 ***Y***，这里的向量 ***X*** 和 ***Y*** 均为伪随机整数向量。第 35 行将图像矩阵 ***P*** 赋给变量 ***A***。第 36～38 行对矩阵 ***A*** 进行行置乱。第 39 行显示行置乱后的图像 ***A***。第 40 行将图像 ***A*** 赋给变量 ***B***。第 41～43 行对矩阵 ***B*** 进行列置乱。第 44 行显示列置乱后的图像 ***B***。

程序段 3-1 运行结果如图 3-2 所示。

在图 3-2 中，对比图 3-2(a)和(b)可知，行置乱对于灰度值相同的列(例如图 3-2(a)中

左边的白色竖条)不起作用,经过一次行置乱和一次列置乱后的图像(见图 3-2(c))能较好地隐藏原始图像信息。



(a) Lena图像　(b) 行置乱后的图像　(c) 列置乱后的图像

图 3-2　程序 3-1 运行结果

2) 单次不重复置乱方法

随机行置乱与列置乱方法中,有些行(或列)会被多次交换位置,而有些行(或列)可能保持不变。例如,对于 3×5 的图像矩阵,若行置乱向量为 $\boldsymbol{X}=[1\ 3\ 2]$,列置乱向量 $\boldsymbol{Y}=[3\ 4\ 4\ 1\ 3]$,则进行行置乱后,图像矩阵不发生变化;进行列置乱后,图像矩阵第 1 列元素被交换 3 次后,仍然处于第 1 列。单次不重复置乱方法可有效避免这些问题。

在单次不重复置乱方法中:

(1) 对于行置乱,借助于超混沌 Lorenz 系统产生长度为 M 的随机数向量 $\boldsymbol{X}$,每个随机数 $X_i\in\{1,2,\cdots,M\}$。然后,在向量 $\boldsymbol{X}$ 中重复的随机数只保留一个。接着,将集合 $\{1,2,\cdots,M\}$ 中没有出现在向量 $\boldsymbol{X}$ 中的元素,按由小到大的顺序排列在向量 $\boldsymbol{X}$ 的末尾。最后,依次将图像矩阵 $\boldsymbol{P}$ 的第 X_i 行与第 X_{M-i+1} 行互换,$i=1,2,\cdots,\lfloor M/2\rfloor$。

(2) 对于列置乱,借助于超混沌 Lorenz 系统产生长度为 N 的随机数向量 $\boldsymbol{Y}$,每个随机数 $Y_i\in\{1,2,\cdots,N\}$。然后,在向量 $\boldsymbol{Y}$ 中重复的随机数只保留一个。接着,将集合 $\{1,2,\cdots,N\}$ 中没有出现在向量 $\boldsymbol{Y}$ 中的元素按由小到大的顺序排列在向量 $\boldsymbol{Y}$ 的末尾。最后,依次将图像矩阵 $\boldsymbol{P}$ 的第 Y_j 列与第 Y_{N-j+1} 列互换,$j=1,2,\cdots,\lfloor N/2\rfloor$。

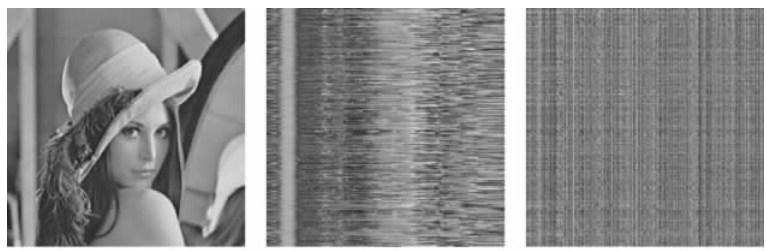
程序段 3-2　单次不重复置乱方法实现行置乱与列置乱

```
clc;clear;
⋮  此处省略的第 2～32 行与程序段 3-1 中的第 2～32 行相同
X = mod(floor((s(1:M) + 100) * 10 ^10),M) + 1;
[～,idx] = unique(X);L = length(idx);X1 = zeros(1,M);
X1(1:length(idx)) = X(sort(idx));
X1(length(idx) + 1:M) = setdiff(1:M,X1);X = X1;

Y = mod(floor((s(M + 1:M + N) + 100) * 10 ^10),N) + 1;
[～,idy] = unique(Y);L = length(idy);Y1 = zeros(1,N);
Y1(1:length(idy)) = Y(sort(idy));
Y1(length(idy) + 1:N) = setdiff(1:N,Y1);Y = Y1;

A = P;
for i = 1:floor(M/2)
   t = A(X(i),:);A(X(i),:) = A(X(M - i + 1),:);A(X(M - i + 1),:) = t;
end
```

```
47  figure(2);imshow(uint8(A));
48  B = A;
49  for j = 1:floor(N/2)
50    t = B(:,Y(j));B(:,Y(j)) = B(:,Y(N - j + 1));B(:,Y(N - j + 1)) = t;
51  end
52  figure(3);imshow(uint8(B));
```

在程序段 3-2 中，第 33 行生成长度为 M 的伪随机数向量 $\boldsymbol{X}$，各个元素的取值 $X_i \in \{1,2,\cdots,M\}$。第 34～35 行借助 unique 函数去掉向量 $\boldsymbol{X}$ 中重复的元素，即使重复的伪随机数只保留一个(第 1 次出现时的那个)。第 36 行将集合$\{1,2,\cdots,M\}$中没有出现在向量 $\boldsymbol{X}$ 中的元素按由小到大的顺序添加到向量 X 的末尾，即产生无重复的随机数向量 X，用于行置乱。第 38～41 行的算法与第 33～36 行的算法原理相同，这里产生无重复的随机数向量 $\boldsymbol{Y}$ 用于列置乱。第 43 行将图像矩阵 $\boldsymbol{P}$ 赋给变量 $\boldsymbol{A}$。第 44～46 行对矩阵 $\boldsymbol{A}$ 进行行置乱，即第 $X(i)$行与第 $X(M-i+1)$行互换位置，$i=1,2,\cdots,\text{floor}(M/2)$。第 47 行显示行置乱后的图像 $\boldsymbol{A}$。第 48 行将图像矩阵 $\boldsymbol{A}$ 赋给变量 $\boldsymbol{B}$。第 49～51 行对矩阵 $\boldsymbol{B}$ 进行列置乱，即第 $Y(j)$列与第 $Y(N-j+1)$列互换位置，$j=1,2,\cdots,\text{floor}(N/2)$。第 52 行显示列置乱后的图像 $\boldsymbol{B}$。

对比图 3-3 和图 3-2 可知，这两种置乱效果相当，均能较好地隐藏原始图像信息，且运算速度快。



(a) Lena图像　(b) 行置乱后的图像　(c) 列置乱后的图像

图 3-3　程序段 3-2 运行结果

2. 二维图像展开成一维向量后的置乱方法

基于二维图像的行置乱或列置乱，是整行或整列元素交换位置。如果图像矩阵的单个元素间交换位置，习惯上，将二维图像展开成一维向量，然后采用随机置乱或不重复置乱方法交换元素位置，最后将置乱后的一维向量还原成二维图像矩阵。二维图像的一次行置乱和列置乱需要随机数 $M+N$ 个，而展开成一维向量后的置乱需要随机数 $M\times N$ 个。

1) 随机置乱方法

将二维图像矩阵 $\boldsymbol{P}$ 按行或列展开为一维向量，记为 $\boldsymbol{A}$。借助于超混沌 Lorenz 系统产生长度为 $M\times N$ 的伪随机序列 X_i，$i=1,2,\cdots,MN$，然后，依次将 $A(i)$与 $A(X_i)$交换位置。随机置乱方法如程序段 3-3 所示。

程序段 3-3　二维图像展成一维向量后的随机置乱

```
1  clc;clear;
2  P = imread('4.2.04.tiff');P = rgb2gray(P);
```

```
3   iptsetpref('imshowborder','tight');
4   figure(1);imshow(P);
5   [M,N] = size(P);P = double(P);
6
7   n = M * N;
⋮   此处省略的第 8～32 行与程序段 3－1 的第 8～32 行相同
33  X = mod(floor((s + 100) * 10 ^ 10),M * N) + 1;
34  A = P(:);
35  for i = 1:M * N
36      t = A(i);A(i) = A(X(i));A(X(i)) = t;
37  end
38  A = reshape(A,M,N);
39  figure(2);imshow(uint8(A));
```

在程序段 3-3 中，第 2～5 行与程序段 3-1 的第 2～5 行相同，用于读入 Lena 图像 $\boldsymbol{P}$ 和获得图像的大小 $M\times N$。第 7 行设置要产生的伪随机序列 s 的长度为 $M\times N$。第 33 行由浮点数类型的伪随机序列 s 生成整数类型的伪随机序列 X，$X_i\in\{1,2,\cdots,MN\}$。第 34 行由图像矩阵 $\boldsymbol{P}$ 生成一维向量 $\boldsymbol{A}$。第 35～37 行对向量 $\boldsymbol{A}$ 进行置乱。第 38 行将置乱后的向量 $\boldsymbol{A}$ 还原为 $M\times N$ 的矩阵。第 39 行显示置乱后的图像矩阵 $\boldsymbol{A}$，如图 3-4(b)所示。

(a) Lena图像　　(b) 置乱后的图像

图 3-4　程序段 3-3 运行结果

对比图 3-2(c)、图 3-3(c)和图 3-4(b)可知，二维图像展开成一维向量后的随机置乱方法效果更好，但是所需要的伪随机数与图像大小相同，图像像素点的交换次数多且运算量较大。

2) 无重复置乱方法

将二维图像矩阵 $\boldsymbol{P}$ 按行或列展开为一维向量，记为 $\boldsymbol{A}$。借助于超混沌 Lorenz 系统产生长度为 $M\times N$ 的伪随机序列 X_i，$i=1,2,\cdots,MN$，然后，X 中重复出现的伪随机数只保留一个(即第一次出现的那个)，将集合$\{1,2,\cdots,MN\}$中没有出现在 X 中的数值按由小到大的顺序添到加 X 的末尾。最后，将 $A(X_i)$与 $A(X_{MN-i+1})$交换位置。无重复置乱方法如程序段 3-4 所示。

程序段 3-4　二维图像展成一维向量后的无重复置乱

```
1   clc;clear;
⋮   此处省略的第 2～32 行与程序段 3-3 的第 2～32 行相同
33  X = mod(floor((s + 100) * 10 ^ 10),M * N) + 1;
34  [～,idx] = unique(X);L = length(idx);X1 = zeros(1,M);
35  X1(1:length(idx)) = X(sort(idx));
36  X1(length(idx) + 1:M * N) = setdiff(1:M * N,X1);X = X1;
```

```
37
38  A = P(:);
39  for i = 1:floor(M * N/2)
40      t = A(X(i));A(X(i)) = A(X(M * N - i + 1));A(X(M * N - i + 1)) = t;
41  end
42  A = reshape(A,M,N);
43  figure(2);imshow(uint8(A));
```

在程序段 3-4 中，第 33 行生成伪随机序列 $X, X_i \in \{1,2,\cdots,MN\}$。第 34～35 行将 X 中重复的元素只保留一个。第 36 行将集合$\{1,2,\cdots,MN\}$中没有出现在 X 中的元素按从小到大的顺序排列在 X 的末尾。第 38 行由图像矩阵 $\boldsymbol{P}$ 生成一维向量 $\boldsymbol{A}$。第 39～41 行对 $\boldsymbol{A}$ 进行置乱。第 42～43 行将 $\boldsymbol{A}$ 还原为矩阵，并显示在图 3-5(b)中。

(a) Lena图像　　(b) 置乱后的图像

图 3-5　程序段 3-4 运行结果

对比图 3-2(c)、图 3-3(c)和图 3-5(b)可知，二维图像展开成一维向量后的无重复置乱方法效果更好，但是所需要的伪随机数与图像大小相同，图像像素点的交换次数多且运算量较大。

3. 借助 2×2 伪随机矩阵进行二维图像置乱

上面介绍的置乱方法都是可逆的。如果直接使用 2×2 的矩阵 $\boldsymbol{T}$，对图像的像素点位置(x_0, y_0)进行变换，得到新的像素点位置(x_1, y_1)，如式(3-1)所示：

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \boldsymbol{T}\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \bmod \begin{bmatrix} M \\ N \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{3-1}$$

由于像素点的坐标位置只可能为整数，所以要求变换矩阵 $\boldsymbol{T}$ 是整数矩阵。著名的 Arnold 矩阵是整数矩阵且其逆矩阵仍然是整数矩阵，常用于图像置乱算法中。对于整数 a 和 b，Arnold 矩阵如式(3-2)所示：

$$\boldsymbol{T} = \begin{bmatrix} 1 & a \\ b & ab+1 \end{bmatrix} \tag{3-2}$$

Arnold 矩阵的逆矩阵如式(3-3)所示：

$$\boldsymbol{T}^{-1} = \begin{bmatrix} ab+1 & -a \\ -b & 1 \end{bmatrix} \tag{3-3}$$

程序段 3-5　Arnold 伪随机矩阵置乱算法

```
1   clc;clear;
2   P = imread('4.2.04.tiff');P = rgb2gray(P);
```

```
3   iptsetpref('imshowborder','tight');
4   figure(1);imshow(P);
5   [M,N] = size(P);P = double(P);
6
7   n = 2 * M * N;
⋮    此处省略的第 8～32 行与程序段 3－1 的第 8～32 行相同
33  X = mod(floor((s + 100) * 10 ^ 10),10 * max(M,N)) + 1;
34  a = reshape(X(1:M * N),M,N);b = reshape(X(M * N + 1:2 * M * N),M,N);
35  A = P;tic;
36  for i = 1:M
37      for j = 1:N
38          k = mod([1 a(i,j);b(i,j) a(i,j) * b(i,j) + 1] * [i;j],[M;N]) + [1;1];
39          t = A(i,j);A(i,j) = A(k(1),k(2));A(k(1),k(2)) = t;
40    end
41  end
42  toc;figure(2);imshow(uint8(A));
43
44  B = A;
45  for i = M: - 1:1
46      for j = N: - 1:1
47          k = mod([1 a(i,j);b(i,j) a(i,j) * b(i,j) + 1] * [i;j],[M;N]) + [1;1];
48          t = B(i,j);B(i,j) = B(k(1),k(2));B(k(1),k(2)) = t;
49    end
50  end
51  figure(3);imshow(uint8(B));
```

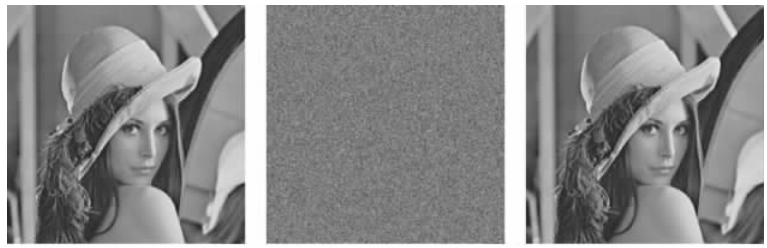
在程序段 3-5 中，第 2～5 行与程序段 3-1 的第 2～5 行相同，用于读入 Lena 图像 $\boldsymbol{P}$ 和获得图像的大小 $M\times N$。第 7 行设置要产生的伪随机序列 s 的长度为 $2MN$。第 33 行由浮点数类型的伪随机序列 s 生成整数类型的伪随机序列 X，$X_i\in\{1,2,\cdots,10MN\}$，长度为 $2MN$。第 34 行由向量 $\boldsymbol{X}$ 生成伪随机矩阵 $\boldsymbol{a}$ 和 $\boldsymbol{b}$，大小均为 $\boldsymbol{M}\times\boldsymbol{N}$。第 35 行由图像矩阵 $\boldsymbol{P}$ 生成一维向量 $\boldsymbol{A}$。第 36～41 行借助于 Arnold 矩阵对图像 $\boldsymbol{A}$ 进行置乱，其中，第 38 行由坐标 (i,j) 计算新的坐标位置向量 $\boldsymbol{k}$。第 39 行将坐标 (i,j) 处的元素与坐标 $(k(1),k(2))$ 处的元素交换。第 42 行显示置乱后的图像矩阵 $\boldsymbol{A}$，如图 3-6(b)所示。第 44 行将置乱后的图像矩阵 $\boldsymbol{A}$ 赋给变量 $\boldsymbol{B}$。第 45～50 行从右至左、从下至上扫描图像 $\boldsymbol{B}$，再次借助 Arnold 矩阵还原图像矩阵 $\boldsymbol{B}$。第 51 行显示被还原的图像，如图 3-6(c)所示。

程序段 3-5 中第 35 行 tic 和第 42 行的 toc 是一对指令，用于测量第 36～41 行的执行时间，在本书使用的计算机上(Intel i7-4720HQ 处理器＋32GB DDR3L 内存＋MATLAB 2015a＋Windows 10 操作系统)，运行时间为 1.4258s。

借助于 Arnold 伪随机矩阵置乱图像的算法运算量比前述的所有置乱方法都要大得多，因此，运算时间相对较长，但置乱效果较好，如图 3-6(b)所示。

从程序段 3-5 的第 44～51 行可知，置乱算法的逆过程仍然使用了 Arnold 矩阵，而不是 Arnold 矩阵的逆矩阵。因此，任意随机矩阵(包括那些不可逆的随机矩阵)都可以用于置乱算法中。只要置乱算法本身是可逆的，就是合法的置乱算法。

下面对 Arnold 矩阵变换方法进行优化：将明文图像 $\boldsymbol{P}$ 展开为一维行向量，记为 $\boldsymbol{A}$，大小为 MN。对向量 $\boldsymbol{A}$ 的任一点坐标位置 $(1,j)$ 进行 Arnold 变换，得新的坐标位置



(a) Lena图像　(b) 置乱后的图像　(c) 还原后的图像

图 3-6　程序段 3-5 运行结果

(p,q)，如式(3-4)所示：

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & a \\ b & ab+1 \end{bmatrix} \begin{bmatrix} 1 \\ j \end{bmatrix} \tag{3-4}$$

所以，

$$p = 1 + aj \tag{3-5}$$

$$q = b + (ab+1)j \tag{3-6}$$

仅考虑上述的式(3-6)可知，可实现通过伪随机变量 a 和 b 实现像素点$(1,j)$和$(1,q)$之间的交换，即不考察 p 的作用。同时，将 $ab+1$ 视为一个新的随机数，仍记为 a，则式(3-6)变为如式(3-7)所示：

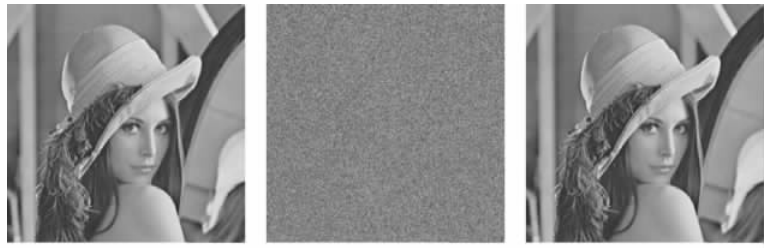
$$q = b + aj \tag{3-7}$$

程序段 3-6　优化后的图像置乱算法

```
1   clc;clear;
⋮   此处省略的第 2～32 行与程序段 3-5 的第 2～32 行相同
33  X = mod(floor((s + 100) * 10 ^ 10),10 * max(M,N)) + 1;
34  a = X(1:M * N);b = X(M * N + 1:2 * M * N);
35  tic;
36  A = P(:);q = mod(b + a. * (1:M * N),M * N) + 1;
37  for j = 1:M * N
38      t = A(j);A(j) = A(q(j));A(q(j)) = t;
39  end
40  A = reshape(A,M,N);
41  toc;figure(2);imshow(uint8(A));
42
43  B = A(:);
44  for j = M * N: - 1:1
45      t = B(j);B(j) = B(q(j));B(q(j)) = t;
46  end
47  B = reshape(B,M,N);
48  figure(3);imshow(uint8(B));
```

在程序段 3-6 中，第 33 行由 s 得到长度为 $2MN$ 的伪随机数向量 $\boldsymbol{X}$，$X_i \in \{1,2,\cdots,10MN\}$。第 34～35 行由 $\boldsymbol{X}$ 得到伪随机数向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$。第 36 行将图像矩阵 $\boldsymbol{P}$ 转化为一维向量 $\boldsymbol{A}$，由向量 $\boldsymbol{a}$ 和 $\boldsymbol{b}$ 得到 q。第 37～39 行对 $\boldsymbol{A}$ 进行置乱。第 40 行将一维向量 $\boldsymbol{A}$ 还原为 $M \times N$ 的图像矩阵。第 41 行显示图像矩阵 $\boldsymbol{A}$，如图 3-7(b)所示。第 43～48 行为置

乱算法的逆过程，还原后的图像如图 3-7(c)所示。



(a) Lena图像　(b) 置乱后的图像　(c) 还原后的图像

图 3-7　程序段 3-6 运行结果

程序段 3-6 中置乱算法的运行时间约为 0.0250s，远远小于程序段 3-5 中置乱算法的运行时间。由图 3-7(b)可知，置乱效果较好。因此，图像置乱算法常常基于其展开成的一维向量进行。单纯的置乱算法，无论算法多么复杂，都是“纯”加密处理，无法对抗选择明文攻击和已知明文攻击。因此，密码学家建议尽可能使用效率高的置乱算法为后续的扩散算法服务，或者在扩散的过程中同步使用置乱算法。

3.3　扩散算法

在图像加密系统中，扩散处理是在不改变像素点位置的条件下，将任一明文像素点的信息隐藏在尽可能多的密文像素点中。本书中，设明文图像为 8b 的灰度图像。最基本的扩散算法如式(3-8)所示：

$$C_i = C_{i-1} \oplus S_i \oplus P_i \tag{3-8}$$

其中，明文图像被展开成一维向量，记为 $\boldsymbol{P}$，相应的密文也为一维向量，记 $\boldsymbol{C}$，$\boldsymbol{S}$ 为密码向量，$i=1,2,\cdots,MN$。初始值 C_0 来自密钥。

展开式(3-8)可得下述关系式：

$$C_1 = C_0 \oplus S_1 \oplus P_1 \tag{3-9}$$

$$C_2 = C_0 \oplus S_1 \oplus S_2 \oplus P_1 \oplus P_2 \tag{3-10}$$

$$C_3 = C_0 \oplus S_1 \oplus S_2 \oplus S_3 \oplus P_1 \oplus P_2 \oplus P_3 \tag{3-11}$$

$$C_4 = C_0 \oplus S_1 \oplus S_2 \oplus S_3 \oplus S_4 \oplus P_1 \oplus P_2 \oplus P_3 \oplus P_4 \tag{3-12}$$

$$C_n = C_0 \oplus S_1 \oplus S_2 \oplus \cdots \oplus S_n \oplus P_1 \oplus P_2 \oplus \cdots \oplus P_n \tag{3-13}$$

由上述关系式可知，明文图像中像素点 P_1 的信息通过异或运算扩散到了全部密文的像素点中，P_2 的信息通过异或运算扩散到了密文的 $C_2 \sim C_{MN}$ 中……P_{MN} 的信息将只隐藏在 C_{MN} 中。因此，按一维向量 $\boldsymbol{P}$ 的正向(i 从 1 到 MN)进行一次式(3-8)所示的运算，扩散效果是有限的。若将得到的 $\boldsymbol{C}$ 向量赋给 $\boldsymbol{P}$，再按一维向量 $\boldsymbol{P}$ 的逆向(i 从 MN 到 1)进行一次式(3-8)的运算，理论上，每个明文像素点的信息都扩散到了密文的每个像素点中。也就是说，采用式(3-8)所示的异或运算进行扩散处理，其图像加密操作至少要循环 2 次。

正向(按 i 从 1 到 MN)的算法与其逆算法如式(3-14)和式(3-15)所示：

$$C_i = C_{i-1} \oplus S_i \oplus P_i \tag{3-14}$$

$$P_i = C_{i-1} \oplus C_i \oplus S_i \tag{3-15}$$

逆向(按 i 从 MN 到 1)的算法与其逆算法如式(3-16)和式(3-17)所示：

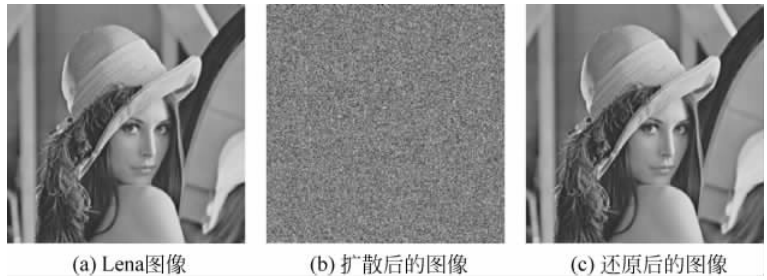
$$C_i = C_{i+1} \oplus S_i \oplus P_i \tag{3-16}$$

$$P_i = C_{i+1} \oplus C_i \oplus S_i \tag{3-17}$$

程序段 3-7　基于异或运算的扩散处理

```
1   clc;clear;
⋮   此处省略的第 2～32 行与程序段 3-5 的第 2～32 行相同
33  S = mod(floor(s * pow2(16)),256);
34  S1 = S(1:M * N);S2 = S(M * N + 1:2 * M * N);B = zeros(M,N);C = zeros(M,N);
35  tic;A = P(:);B0 = 0;B(1) = bitxor(bitxor(B0,S1(1)),A(1));
36  for i = 2:M * N
37      B(i) = bitxor(bitxor(B(i - 1),S1(i)),A(i));
38  end
39  C0 = 0;C(M * N) = bitxor(bitxor(C0,S2(M * N)),B(M * N));
40  for i = M * N - 1: - 1:1
41      C(i) = bitxor(bitxor(C(i + 1),S2(i)),B(i));
42  end
43  C = reshape(C,M,N);toc;
44  figure(2);imshow(uint8(C));
45
46  A = C(:);D = zeros(M,N);E = zeros(M,N);
47  D0 = 0;D(M * N) = bitxor(bitxor(D0,S2(M * N)),C(M * N));
48  for i = M * N - 1: - 1:1
49      D(i) = bitxor(bitxor(C(i + 1),S2(i)),C(i));
50  end
51  E0 = 0;E(1) = bitxor(bitxor(E0,S1(1)),D(1));
52  for i = 2:M * N
53      E(i) = bitxor(bitxor(D(i - 1),S1(i)),D(i));
54  end
55  E = reshape(E,M,N);figure(3);imshow(uint8(E));
```

在程序段 3-7 中，第 34 行的 S1 和 S2 分别用作正向扩散或逆向扩散的伪随机序列，***B*** 保存正向扩散算法得到的中间结果，***C*** 保存密文。第 35～38 行为正向扩散算法，如式(3-14)所示。第 39～42 行为逆向扩散算法，如式(3-16)所示。第 43 行将一维向量 ***C*** 转化为 $M \times N$ 矩阵。第 44 行显示密文图像，如图 3-8(b)所示。第 46 行将密文图像 ***C*** 展成一维向量赋给变量 ***A***，***D*** 保存中间结果，***E*** 保存还原后的图像。第 47～50 行为正向扩



(a) Lena图像　(b) 扩散后的图像　(c) 还原后的图像

图 3-8　程序段 3-7 运行结果

散算法的逆过程，如式(3-15)所示。第 51～54 行为逆向扩散算法的逆过程，如式(3-17)所示。第 55 行将 $\boldsymbol{E}$ 转化为 $M\times N$ 矩阵，并显示图像 $\boldsymbol{E}$，如图 3-8(c)所示。

程序段 3-7 中第 35～43 行(即基于异或运算的扩散算法)的运行时间约为 0.7138s。

除了异或运算外，更常用的扩散算法是加取模运算，如式(3-18)所示：

$$C_i = (C_{i-1} + S_i + P_i) \bmod 256 \tag{3-18}$$

展开式(3-18)可得

$$C_n = (C_0 + S_1 + S_2 + \cdots + S_n + P_1 + P_2 + \cdots + P_n) \bmod 256 \tag{3-19}$$

在加取模运算中，明文像素点 P_i 的信息只能“隐藏”到 $C_i \sim C_{MN}$ 中，所以，这种方法需要循环 2 次才能实现任意明文像素点的信息扩散到整个密文图像中。

正向(按 i 从 1 到 MN)的算法与其逆算法如式(3-20)和式(3-21)所示：

$$C_i = (C_{i-1} + S_i + P_i) \bmod 256 \tag{3-20}$$

$$P_i = (2\times 256 + C_i - C_{i-1} - S_i) \bmod 256 \tag{3-21}$$

逆向(按 i 从 MN 到 1)的算法与其逆算法如式(3-22)和式(3-23)所示。

$$C_i = (C_{i+1} + S_i + P_i) \bmod 256 \tag{3-22}$$

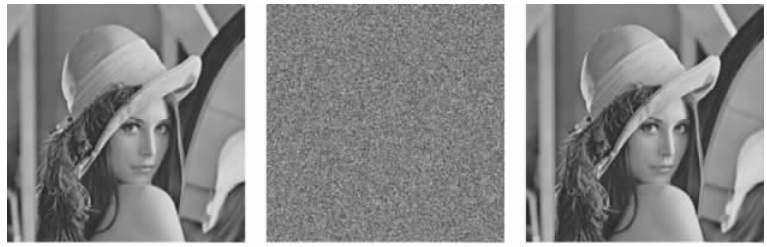
$$P_i = (2\times 256 + C_i - C_{i+1} - S_i) \bmod 256 \tag{3-23}$$

程序段 3-8　基于加取模运算的扩散算法

```
1   clc;clear;close all;
⋮   此处省略的第 2～33 行与程序段 3-7 的第 2～33 行相同
34  S1 = S(1:M * N);S2 = S(M * N + 1:2 * M * N);B = zeros(M,N);C = zeros(M,N);
35  tic;A = P(:);B0 = 0;B(1) = mod(B0 + S1(1) + A(1),256);
36  for i = 2:M * N
37      B(i) = mod(B(i - 1) + S1(i) + A(i),256);
38  end
39  C0 = 0;C(M * N) = mod(C0 + S2(M * N) + B(M * N),256);
40  for i = M * N - 1: - 1:1
41      C(i) = mod(C(i + 1) + S2(i) + B(i),256);
42  end
43  C = reshape(C,M,N);toc;
44  figure(2);imshow(uint8(C));
45
46  A = C(:);D = zeros(M,N);E = zeros(M,N);
47  D0 = 0;D(M * N) = mod(256 * 2 + C(M * N) - D0 - S2(M * N),256);
48  for i = M * N - 1: - 1:1
49      D(i) = mod(256 * 2 + C(i) - C(i + 1) - S2(i),256);
50  end
51  E0 = 0;E(1) = mod(256 * 2 + D(1) - E0 - S1(1),256);
52  for i = 2:M * N
53      E(i) = mod(256 * 2 + D(i) - D(i - 1) - S1(i),256);
54  end
55  E = reshape(E,M,N);figure(3);imshow(uint8(E));
```

在程序段 3-8 中，第 34 行的 S1 和 S2 保存正向算法与逆向算法的伪随机数向量，$\boldsymbol{B}$ 保存中间结果，$\boldsymbol{C}$ 保存密文。第 35～38 行为正向算法，由向量 $\boldsymbol{A}$ 得到向量 $\boldsymbol{B}$。第 39～42 行为逆向算法，由向量 $\boldsymbol{B}$ 得到向量 $\boldsymbol{C}$。第 43 行将向量 $\boldsymbol{C}$ 转化为 $M\times N$ 矩阵。第 44 行显

示密文图像 $\boldsymbol{C}$，如图 3-9(b)所示。第 46～55 行为扩散算法的逆过程。第 46 行将密文图像 $\boldsymbol{C}$ 转化为一维向量保存在变量 $\boldsymbol{A}$ 中，变量 $\boldsymbol{D}$ 保存中间结果，变量 $\boldsymbol{E}$ 保存还原后的图像。第 47～50 行为正向算法的逆算法。第 51～54 行为逆向算法的逆算法。第 55 行显示还原后的图像，如图 3-9(c)所示。



(a) Lena图像　(b) 扩散后的图像　(c) 还原后的图像

图 3-9　程序段 3-8 运行结果

程序段 3-8 中第 35～43 行(即基于加取模运算的扩散算法)的执行时间约为 0.0206s，远远小于基于异或运算的扩散算法的执行时间(0.7138s)，这是因为 MATLAB 软件中位运算能力弱，在借助 C 语言时，两者的执行时间应相当。

在扩散算法中，一个显著的改良处理是在式(3-8)或式(3-18)中添加循环移位操作，如式(3-24)和式(3-25)所示：

$$C_i = (C_{i-1} \oplus S_i \oplus P_i) <<< \mathrm{LSB}_3(C_{i-1}) \tag{3-24}$$

$$C_i = (C_{i-1} + S_i + P_i) \bmod 256 <<< \mathrm{LSB}_3(C_{i-1}) \tag{3-25}$$

其中，LSB_3 表示取数据的最低 3 位，由于本书使用了 8b 的灰度图像，因此每个像素点是 8b 的，任一数据的低 3 位取值范围为 0～7，是一个像素点数据循环移位的有效范围。如果是 l 位的灰度图像，那应取数据的最低 $\log_2 l$ 位或者任意的 $\log_2 l$ 位。除了循环左移外，还可以使用循环右移。如果加密算法中使用了循环左移，则解密算法中应使用循环右移；同样地，如果加密算法中使用了循环右移，则解密算法中应使用循环左移。

不失一般性，下面以式(3-25)为例进行算法设计，如程序段 3-9 所示。

程序段 3-9　基于加取模和循环左移运算的扩散算法

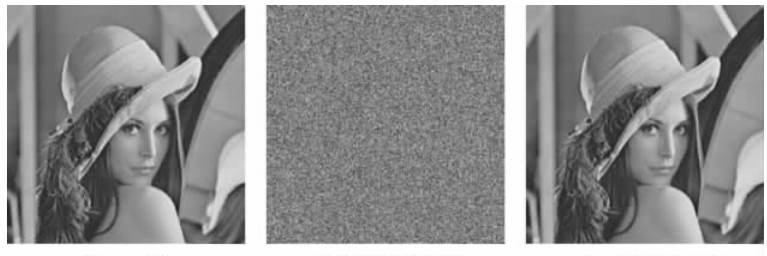
```
1   clc;clear;close all;
⋮   此处省略的第 2～32 行与程序段 3-7 的第 2～32 行相同
33  S = mod(floor(s * pow2(16)),256);
34  S1 = S(1:M * N);S2 = S(M * N + 1:2 * M * N);B = zeros(M,N);C = zeros(M,N);
35  tic;A = P(:);B0 = 0;B(1) = mod(B0 + S1(1) + A(1),256);
36  B(1) = BitCircShift(B(1),mod(B0,8));
37  for i = 2:M * N
38      B(i) = mod(B(i - 1) + S1(i) + A(i),256);
39      B(i) = BitCircShift(B(i),mod(B(i - 1),8));
40  end
41  C0 = 0;C(M * N) = mod(C0 + S2(M * N) + B(M * N),256);
42  C(M * N) = BitCircShift(C(M * N),mod(C0,8));
43  for i = M * N - 1: - 1:1
44      C(i) = mod(C(i + 1) + S2(i) + B(i),256);
45      C(i) = BitCircShift(C(i),mod(C(i + 1),8));
```

```
46  end
47  C = reshape(C,M,N);toc;
48  figure(2);imshow(uint8(C));
49
50  A = C(:);D = zeros(M,N);E = zeros(M,N);
51  A0 = 0;A(M * N) = BitCircShift(A(M * N), - mod(A0,8));
52  D(M * N) = mod(256 * 2 + A(M * N) - A0 - S2(M * N),256);
53  for i = M * N - 1: - 1:1
54      D(i) = BitCircShift(A(i), - mod(A(i + 1),8));
55      D(i) = mod(256 * 2 + D(i) - A(i + 1) - S2(i),256);
56  end
57  E0 = 0;E(1) = BitCircShift(D(1), - mod(E0,8));
58  E(1) = mod(256 * 2 + E(1) - E0 - S1(1),256);
59  for i = 2:M * N
60      E(i) = BitCircShift(D(i), - mod(D(i - 1),8));
61      E(i) = mod(256 * 2 + E(i) - D(i - 1) - S1(i),256);
62  end
63  E = reshape(E,M,N);figure(3);imshow(uint8(E));
```

程序段 3-10　循环移位函数 BitCircShift

```
1   function y = BitCircShift(x,k)
2   if abs(k)> 7 || k == 0
3       y = x;return;
4   end
5   if k > 0
6       t1 = mod(x * pow2(k),256);t2 = floor(x/pow2(8 - k));
7   else
8       t1 = floor(x * pow2(k));t2 = mod(x,pow2( - k)) * pow2(8 + k);
9   end
10  y = t1 + t2;
11  end
```

对比程序段 3-8 和程序段 3-9 可知，在正向扩散算法中添加了第 36、39 行，在逆向扩散算法中添加了第 42、45 行，这 4 行调用自定义函数 BitCircShift(如程序段 3-10 所示)，对数据点进行循环左移操作，如式(3-25)所示，扩散算法得到的图像如图 3-10(b)所示。第 50～63 行为扩散算法的逆运算，由于扩散算法中使用循环左移，所以这里第 51、55、57 和 60 行采用了循环右移，扩散算法的逆运算后还原出来的图像如图 3-10(c)所示。



(a) Lena图像　(b) 扩散后的图像　(c) 还原后的图像

图 3-10　程序段 3-9 运行结果

在程序段 3-10 的自定义函数 BitCircShift 中，输入参数为 x 和 k，输出参数为 y，实现的功能为：如果 $0<k<8$，则 $y=x<<<k$；如果 $-8<k<0$，则 $y=x>>>(-k)$。

由于 MATLAB 不支持循环移位操作，所以添加了循环移位操作后的程序（第 35～47 行）运算速度较慢，其执行时间约为 2.4566s。

除了上述常用的扩散算法外，还有借助于有限域 GF(2^n)的扩散算法，下面介绍 $n=4$ 和 $n=8$ 的情况下的扩散算法。

1. 有限域 GF(2^4)下的扩散算法

在 GF(2^4)中取既约多项式 $m(x)=x^4+x+1$，则其加法和乘法运算规律如图 3-11 所示。

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

(a) 加法

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

(b) 乘法

图 3-11　GF(2^4)域（既约多项式 $m(x)=x^4+x+1$）中的加法和乘法运算

由图 3-11(a)可知,可直接在扩散算法中应用 GF(2^4)域的加法运算。由图 3-11(b)可知,由于 0 作为乘数时有信息的损失,所以不能直接将 GF(2^4)域的乘法运算应用于扩散算法中。一般地,在扩散算法中使用 GF(17)域的乘法运算,如图 3-12 所示。

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

图 3-12　GF(17)域中的乘法运算

对于使用如图 3-11(a)所示的 GF(2^4)域进行扩散的情况,正向扩散时,借助于式(3-26),正向扩散的逆运算如式(3-27)所示;逆向扩散时,借助于式(3-28),逆向扩散的逆运算如式(3-29)所示。

$$\begin{cases} C_{i,\mathrm{H}} = C_{i-1,\mathrm{H}} + S_{i,\mathrm{H}} + P_{i,\mathrm{H}}, \quad C_{i,\mathrm{L}} = C_{i-1,\mathrm{L}} + S_{i,\mathrm{L}} + P_{i,\mathrm{L}} \\ C_i = (C_{i,\mathrm{H}} \times 16 + C_{i,\mathrm{L}}) \end{cases} \tag{3-26}$$

$$\begin{cases} P_{i,\mathrm{H}} = C_{i,\mathrm{H}} - C_{i-1,\mathrm{H}} - S_{i,\mathrm{H}}, \quad P_{i,\mathrm{L}} = C_{i,\mathrm{L}} - C_{i-1,\mathrm{L}} - S_{i,\mathrm{L}} \\ P_i = (P_{i,\mathrm{H}} \times 16 + P_{i,\mathrm{L}}) \end{cases} \tag{3-27}$$

$$\begin{cases} C_{i,\mathrm{H}} = C_{i+1,\mathrm{H}} + S_{i,\mathrm{H}} + P_{i,\mathrm{H}}, \quad C_{i,\mathrm{L}} = C_{i+1,\mathrm{L}} + S_{i,\mathrm{L}} + P_{i,\mathrm{L}} \\ C_i = (C_{i,\mathrm{H}} \times 16 + C_{i,\mathrm{L}}) \end{cases} \tag{3-28}$$

$$\begin{cases} P_{i,\mathrm{H}} = C_{i,\mathrm{H}} - C_{i+1,\mathrm{H}} - S_{i,\mathrm{H}}, \quad P_{i,\mathrm{L}} = C_{i,\mathrm{L}} - C_{i+1,\mathrm{L}} - S_{i,\mathrm{L}} \\ P_i = (P_{i,\mathrm{H}} \times 16 + P_{i,\mathrm{L}}) \end{cases} \tag{3-29}$$

式(3-26)至式(3-29)中的“+”和“−”均为 GF(2^4)上的加法和减法。

对于使用如图 3-12 所示的 GF(17)域进行扩散的情况,正向扩散时,借助于式(3-30),正向扩散的逆运算如式(3-31)所示;逆向扩散时,借助于式(3-32),逆向扩散的逆运算如式(3-33)所示。

$$\begin{cases} C_{i,\mathrm{H}} = C_{i-1,\mathrm{H}} \times S_{i,\mathrm{H}} \times P_{i,\mathrm{H}}, \quad C_{i,\mathrm{L}} = C_{i-1,\mathrm{L}} \times S_{i,\mathrm{L}} \times P_{i,\mathrm{L}} \\ C_i = (C_{i,\mathrm{H}} \times 16 + C_{i,\mathrm{L}}) \end{cases} \tag{3-30}$$

$$\begin{cases} P_{i,\mathrm{H}} = C_{i,\mathrm{H}} \div C_{i-1,\mathrm{H}} \div S_{i,\mathrm{H}}, \quad P_{i,\mathrm{L}} = C_{i,\mathrm{L}} \div C_{i-1,\mathrm{L}} \div S_{i,\mathrm{L}} \\ P_i = (P_{i,\mathrm{H}} \times 16 + P_{i,\mathrm{L}}) \end{cases} \tag{3-31}$$

$$\begin{cases} C_{i,\mathrm{H}} = C_{i+1,\mathrm{H}} \times S_{i,\mathrm{H}} \times P_{i,\mathrm{H}}, \quad C_{i,\mathrm{L}} = C_{i+1,\mathrm{L}} \times S_{i,\mathrm{L}} \times P_{i,\mathrm{L}} \\ C_i = (C_{i,\mathrm{H}} \times 16 + C_{i,\mathrm{L}}) \end{cases} \tag{3-32}$$

$$\begin{cases} P_{i,\mathrm{H}} = C_{i,\mathrm{H}} \div C_{i+1,\mathrm{H}} \div S_{i,\mathrm{H}}, \quad P_{i,\mathrm{L}} = C_{i,\mathrm{L}} \div C_{i+1,\mathrm{L}} \div S_{i,\mathrm{L}} \\ P_i = (P_{i,\mathrm{H}} \times 16 + P_{i,\mathrm{L}}) \end{cases} \tag{3-33}$$

式(3-30)至式(3-33)中的“×”和“÷”均为 GF(2^4)上的加法和减法。

式(3-26)至式(3-33)中,下标 H 表示数据的高 4 位,下标 L 表示数据的低 4 位。可以在式(3-26)至式(3-33)中添加循环移位操作。

程序段 3-11　基于 GF(2^4)域的加法运算的扩散算法

```
clc;clear;close all;
⋮  此处省略的第 2～32 行与程序段 3-7 的第 2～32 行相同
TBL = zeros(16,16);a = zeros(1,4);b = a;m = [1 0 0 1 1];
for i = 0:pow2(4) - 1
    a(1) = floor(i/pow2(3));a(2) = mod(floor(i/pow2(2)),2);
    a(3) = mod(floor(i/pow2(1)),2);a(4) = mod(i,2);
    for j = 0:pow2(4) - 1
        b(1) = floor(j/pow2(3));b(2) = mod(floor(j/pow2(2)),2);
        b(3) = mod(floor(j/pow2(1)),2);b(4) = mod(j,2);
        t = mod(a + b,2);[～,r] = deconv(t,m);r = mod(r,2);
        v = r(4) + r(3) * 2 + r(2) * pow2(2) + r(1) * pow2(3);
        TBL(i + 1,j + 1) = v;
    end
end

S = mod(floor(s * pow2(16)),256);
S1 = S(1:M * N);S2 = S(M * N + 1:2 * M * N);B = zeros(M,N);C = zeros(M,N);
tic;A = P(:);B0 = 0;B(1) = LookUpGF2p4(B0,S1(1),A(1),TBL);
for i = 2:M * N
    B(i) = LookUpGF2p4(B(i - 1),S1(i),A(i),TBL);
end
C0 = 0;C(M * N) = LookUpGF2p4(C0,S2(M * N),B(M * N),TBL);
for i = M * N - 1: - 1:1
    C(i) = LookUpGF2p4(C(i + 1),S2(i),B(i),TBL);
end
C = reshape(C,M,N);toc;
figure(2);imshow(uint8(C));

A = C(:);D = zeros(M,N);E = zeros(M,N);
A0 = 0;D(M * N) = LookUpGF2p4Ex(A(M * N),A0,S2(M * N),TBL);
for i = M * N - 1: - 1:1
    D(i) = LookUpGF2p4Ex(A(i),A(i + 1),S2(i),TBL);
```

```
63  end
64  E0 = 0;E(1) = LookUpGF2p4Ex(D(1),E0,S1(1),TBL);
65  for i = 2:M * N
66      E(i) = LookUpGF2p4Ex(D(i),D(i - 1),S1(i),TBL);
67  end
68  E = reshape(E,M,N);figure(3);imshow(uint8(E));
```

在程序段 3-11 中，第 33～44 行生成 GF(2^4)域的加法运算查找表 TBL，其中 m＝[1 0 0 1 1]表示既约多项式 $m(x)=x^4+x+1$，对于加法运算，第 40 行的“[～,r]＝deconv(t,m);r＝mod(r,2);”可改为“r＝mod(t,2);”，既约多项式主要用于乘法运算中。

对比程序段 3-8 和程序段 3-11 可知，在正向扩散算法中，程序段 3-8 中第 35 行计算 B(1)的值和第 37 行计算 B(i)的值的方法，替换为程序段 3-11 中第 48 行和第 50 行用查找表计算 B(1)和 B(i)的值；在逆向扩散算法中，程序段 3-8 中第 39 行计算 C(M * N)的值和第 41 行计算 C(i)的值的方法，替换为程序段 3-11 中第 52 行和第 54 行用查找表计算 C(M * N)和 C(i)的值。这里的自定义函数 LookUpGF2p4 和 LookUpGF2p4Ex 分别如程序段 3-12 和程序段 3-13 所示。

程序段 3-11 运行后，扩散得到的图像如图 3-13(b)所示；扩散的逆过程(如第 59～68 行所示)还原出来的图像如图 3-13(c)所示。

程序段 3-12　基于 GF(2^4)域的加法运算函数

```
1   function y = LookUpGF2p4(x0,x1,x2,TBL)
2   x0H = floor(x0/16);x0L = mod(x0,16);
3   x1H = floor(x1/16);x1L = mod(x1,16);
4   x2H = floor(x2/16);x2L = mod(x2,16);
5   t1 = TBL(TBL(x0L + 1,x1L + 1) + 1,x2L + 1);
6   t2 = TBL(TBL(x0H + 1,x1H + 1) + 1,x2H + 1);
7   y = t2 * 16 + t1;
8   end
```

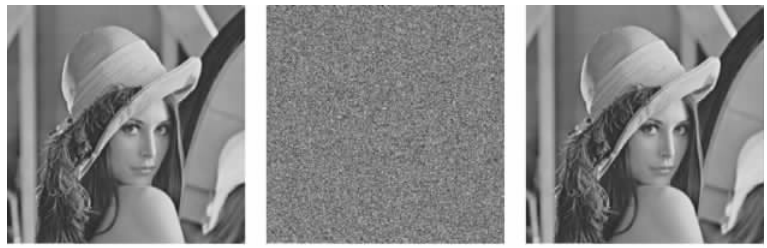
程序段 3-12 实现在 GF(2^4)上 y＝x0＋x1＋x2 的运算。TBL 为 GF(2^4)域加法运算表，如图 3-11(a)所示。

程序段 3-13　基于 GF(2^4)域的减法运算函数

```
1   function y = LookUpGF2p4Ex(x0,x1,x2,TBL)
2   x0H = floor(x0/16);x0L = mod(x0,16);
3   x1H = floor(x1/16);x1L = mod(x1,16);
4   x2H = floor(x2/16);x2L = mod(x2,16);
5   k1 = TBL(x1L + 1,x2L + 1) + 1;
6   k2 = TBL(x1H + 1,x2H + 1) + 1;
7   t1 = find(TBL(k1,:) == x0L) - 1;
8   t2 = find(TBL(k2,:) == x0H) - 1;
9   y = t2 * 16 + t1;
10  end
```

程序段 3-13 实现在 GF(2^4)上 y＝x0－x1－x2 的运算。TBL 为 GF(2^4)域加法运算表，如图 3-11(a)所示。

程序段 3-11 中使用查找表方法，运算速度比程序段 3-9 更快，扩散过程(第 48～56



(a) Lena图像　(b) 扩散后的图像　(c) 还原后的图像

图 3-13　程序段 3-11 运行结果

行)的运行时间约为 1.3096s。

程序段 3-14　基于 GF(17)域的乘法运算的扩散算法

```
1   clc;clear;close all;
⋮   此处省略的第 2～32 行与程序段 3-7 的第 2～32 行相同
33  TBL = mod((transpose(0:16) * (0:16)),17);
34
35  S = mod(floor(s * pow2(16)),256);
36  S1 = S(1:M * N);S2 = S(M * N + 1:2 * M * N);B = zeros(M,N);C = zeros(M,N);
37  tic;A = P(:);B0 = 0;B(1) = LookUpGF17(B0,S1(1),A(1),TBL);
38  for i = 2:M * N
39      B(i) = LookUpGF17(B(i - 1),S1(i),A(i),TBL);
40  end
41  C0 = 0;C(M * N) = LookUpGF17(C0,S2(M * N),B(M * N),TBL);
42  for i = M * N - 1: - 1:1
43      C(i) = LookUpGF17(C(i + 1),S2(i),B(i),TBL);
44  end
45  C = reshape(C,M,N);toc;
46  figure(2);imshow(uint8(C));
47
48  A = C(:);D = zeros(M,N);E = zeros(M,N);
49  A0 = 0;D(M * N) = LookUpGF17Ex(A(M * N),A0,S2(M * N),TBL);
50  for i = M * N - 1: - 1:1
51      D(i) = LookUpGF17Ex(A(i),A(i + 1),S2(i),TBL);
52  end
53  E0 = 0;E(1) = LookUpGF17Ex(D(1),E0,S1(1),TBL);
54  for i = 2:M * N
55      E(i) = LookUpGF17Ex(D(i),D(i - 1),S1(i),TBL);
56  end
57  E = reshape(E,M,N);figure(3);imshow(uint8(E));
```

在程序段 3-14 中,第 33 行生成 GF(17)的乘法运算查找表,如图 3-12 所示。对比程序段 3-11 和程序段 3-14 可知,程序段 3-11 中使用 LookUpGF2p4 的语句(第 48、50、52、54 行),在程序段 3-14 中使用了 LookUpGF17 函数(第 37、39、41、43 行);程序段 3-11 中使用 LookUpGF2p4Ex 的语句(第 60、62、64、66 行),在程序段 3-14 中使用了 LookUpGF17Ex 函数(第 49、51、53、55 行)。除这些调用的函数发生变化外,程序段 3-11 的第 46～68 行与程序段 3-14 的第 35～57 行相同。其中,自定义函数 LookUpGF17 和

LookUpGF17Ex 如程序段 3-15 和程序段 3-16 所示。

程序段 3-14 将 Lena 图像扩散后的结果如图 3-14(b)所示，经扩散算法的逆过程还原后的图像如图 3-14(c)所示。

程序段 3-15　基于 GF(17)域的乘法运算函数

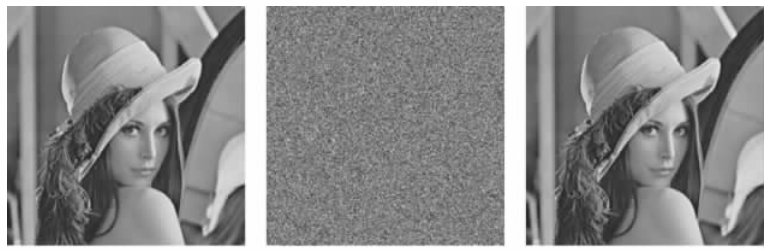
```
function y = LookUpGF17(x0,x1,x2,TBL)
x0H = floor(x0/16) + 1;x0L = mod(x0,16) + 1;
x1H = floor(x1/16) + 1;x1L = mod(x1,16) + 1;
x2H = floor(x2/16) + 1;x2L = mod(x2,16) + 1;
t1 = TBL(TBL(x0L + 1,x1L + 1) + 1,x2L + 1);
t2 = TBL(TBL(x0H + 1,x1H + 1) + 1,x2H + 1);
y = (t2 - 1) * 16 + (t1 - 1);
end
```

程序段 3-15 实现在 GF(17)域上 y＝x0×x1×x2 的运算。TBL 为 GF(17)域乘法运算表，如图 3-12 所示。在 GF(17)域乘法中，输入图像像素点的 8b 灰度值(0～255)被分为高 4 位和低 4 位，分别用含有 H 和 L 的变量名保存。高 4 位和低 4 位的取值均为 0～15，在乘法运算(即查表处理)时，将它们转化为 1～16，查表结果减去 1 还原为真实的高 4 位或低 4 位值。

程序段 3-16　基于 GF(17)域的除法运算函数

```
function y = LookUpGF17Ex(x0,x1,x2,TBL)
    x0H = floor(x0/16) + 1;x0L = mod(x0,16) + 1;
    x1H = floor(x1/16) + 1;x1L = mod(x1,16) + 1;
    x2H = floor(x2/16) + 1;x2L = mod(x2,16) + 1;
    k1 = TBL(x1L + 1,x2L + 1) + 1;
    k2 = TBL(x1H + 1,x2H + 1) + 1;
    t1 = find(TBL(k1,:) = = x0L) - 1;
    t2 = find(TBL(k2,:) = = x0H) - 1;
    y = (t2 - 1) * 16 + (t1 - 1);
    end
```

程序段 3-16 实现在 GF(17)域上 y＝x0÷x1÷x2 的运算。TBL 为 GF(17)域乘法运算表，如图 3-12 所示。在 GF(17)域乘法中，输入图像像素点的 8b 灰度值(0～255)被分为高 4 位和低 4 位，分别用含有 H 和 L 的变量名保存。高 4 位和低 4 位的取值均为 0～15，在除法运算(即查表处理)时，将它们转化为 1～16，查表结果减去 1 还原为真实的高 4 位或低 4 位值。



(a) Lena图像　(b) 扩散后的图像　(c) 还原后的图像

图 3-14　程序段 3-14 运行结果

程序段 3-14 使用查找表方法实现了 GF(17)域的乘法运算，运算速度也比较快，扩散算法(第 37～45 行)的执行时间约为 1.2057s。由图 3-14(b)可知，扩散后的图像的视觉效果较好。

2. 有限域 GF(2^8)下的扩散算法

在 GF(2^8)中，若取既约多项式 $m(x)=x^8+x^4+x^3+x+1$，则其加法和乘法运算表由程序段 3-17 得到，一般地，GF(2^8)域乘法运算不能直接应用于扩散算法，而是借助于 GF(257)域的乘法运算表(由程序段 3-18 得到)进行扩散操作。

程序段 3-17　GF(2^8)域加法和乘法运算表

```
1   function [T1,T2] = GF2p8Table()
2   a = zeros(1,8);b = zeros(1,8);m = [1 0 0 0 1 1 0 1 1];
3   AM = zeros(256,256);PM = zeros(256,256);
4   for i = 0:pow2(8) - 1
5       for j = 1:8
6         a(j) = mod(floor(i/pow2(8 - j)),2);
7       end
8       for j = 0:pow2(8) - 1
9         for k = 1:8
10          b(k) = mod(floor(j/pow2(8 - k)),2);
11        end
12        t = mod(a + b,2);r = mod(t,2);v = sum(r. * pow2(7: - 1:0));
13        AM(i + 1,j + 1) = v;
14    end
15  end
16  T1 = AM;
17
18  for i = 0:pow2(8) - 1
19      for j = 1:8
20        a(j) = mod(floor(i/pow2(8 - j)),2);
21      end
22      for j = 0:pow2(8) - 1
23        for k = 1:8
24          b(k) = mod(floor(j/pow2(8 - k)),2);
25        end
26        t = conv(a,b);t = mod(t,2);[～,r] = deconv(t,m);
27        r = mod(r,2);v = sum(r(8:15). * pow2(7: - 1:0));PM(i + 1,j + 1) = v;
28    end
29  end
30  T2 = PM;
31  end
```

在程序段 3-17 中，第 2～16 行生成 GF(2^8)域的加法运算表，保存在变量 T1 中。第 18～30 行生成 GF(2^8)域的乘法运算表，保存在变量 T2 中。

程序段 3-18　GF(257)域乘法运算表

```
1   function T = GF257Table()
2   T = mod(transpose(0:256) * (0:256),257);
3   end
```

在程序段 3-18 中，第 2 行生成 GF(257)域的乘法运算表。

借助于 GF(2^8)域加法运算表实现的加法运算和减法运算分别如程序段 3-19 和程序段 3-20 所示，借助于 GF(257)域乘法运算表实现的乘法运算和除法运算分别如程序段 3-21 和程序段 3-22 所示。

程序段 3-19　基于 GF(2^8)域的加法运算函数

```
1    function y = LookUpGF2p8(x0,x1,x2,TBL)
2    y = TBL(TBL(x0 + 1,x1 + 1) + 1,x2 + 1);
3    end
```

程序段 3-19 实现在 GF(2^8)域上 y＝x0＋x1＋x2 的运算。TBL 为 GF(2^8)域加法运算表，由程序段 3-17 生成。

程序段 3-20　基于 GF(2^8)域的减法运算函数

```
1    function y = LookUpGF2p8Ex(x0,x1,x2,TBL)
2    t = TBL(x1 + 1,x2 + 1) + 1;y = find(TBL(t,:) = = x0) - 1;
3    end
```

程序段 3-20 实现在 GF(2^8)域上 y＝x0－x1－x2 的运算。TBL 为 GF(2^8)域加法运算表，由程序段 3-17 生成。

程序段 3-21　基于 GF(257)域的乘法运算函数

```
1    function y = LookUpGF257(x0,x1,x2,TBL)
2    y = TBL(TBL(x0 + 2,x1 + 2) + 1,x2 + 2) - 1;
3    end
```

程序段 3-21 实现在 GF(257)域上 y＝x0×x1×x2 的运算。TBL 为 GF(257)域乘法运算表，由程序段 3-18 生成。

程序段 3-22　基于 GF(257)域的除法运算函数

```
1    function y = LookUpGF257Ex(x0,x1,x2,TBL)
2    t = TBL(x1 + 2,x2 + 2) + 1;y = find(TBL(t,:) = = (x0 + 1)) - 2;
3    end
```

程序段 3-22 实现在 GF(257)域上 y＝x0÷x1÷x2 的运算。TBL 为 GF(257)域乘法运算表，由程序段 3-18 生成。

下面讨论基于 GF(2^8)域加法运算和基于 GF(257)域乘法运算的扩散算法，在这些算法中可以添加以循环移位操作。

3. 基于 GF(2^8)域加法运算的扩散算法

新建程序如程序段 3-14 所示，然后做如下修改：

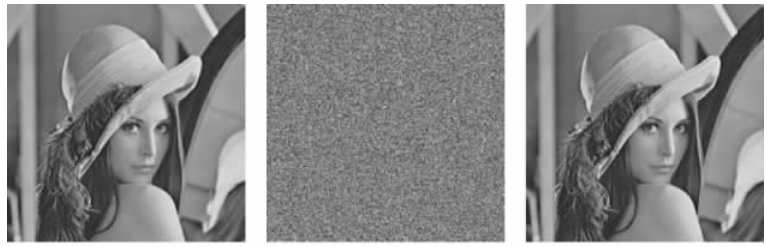
(1) 将第 33 行“TBL＝mod((transpose(0:16) * (0:16)),17);”改为“TBL＝GF2p8Table();”。

(2) 将第 37、39、41、43 行的函数“LookUpGF17”改为“LookUpGF2p8”。

(3) 将第 49、51、53、55 行的函数“LookUpGF17Ex”改为“LookUpGF2p8Ex”。

程序运行结果如图 3-15 所示，扩散算法(程序的第 37～45 行，可参考程序段 3-14)运

行时间约为 0.6822s。



(a) Lena图像　(b) 扩散后的图像　(c) 还原后的图像

图 3-15　基于 GF(2^8)域加法的扩散算法程序运行结果

4. 基于 GF(257)域乘法运算的扩散算法

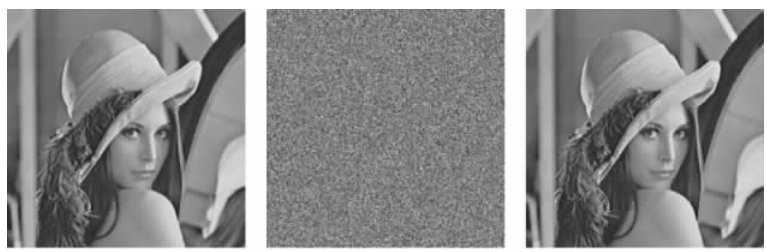
新建程序如程序段 3-14 所示，然后做如下修改：

(1) 将第 33 行“TBL = mod((transpose(0:16) * (0:16)),17);”改为“TBL = GF257Table();”。

(2) 将第 37、39、41、43 行的函数“LookUpGF17”改为“LookUpGF257”。

(3) 将第 49、51、53、55 行的函数“LookUpGF17Ex”改为“LookUpGF257Ex”。

程序运行结果如图 3-16 所示，扩散算法(程序的第 37～45 行，可参考程序段 3-14)运行时间约为 0.7020s。



(a) Lena图像　(b) 扩散后的图像　(c) 还原后的图像

图 3-16　基于 GF(257)域乘法的扩散算法程序运行结果

3.4　本章小结

本章介绍了基于混沌系统的图像加密方案与解密方案的常规结构，然后重点介绍了常用的置乱算法，即二维图像直接行置乱或列置乱、二维图像展开成一维向量后的置乱方法以及借助二阶随机方阵进行图像置乱，最后详细讨论了常规的扩散算法，即基于异或运算的扩散、基于加取模运算的扩散、添加了循环移位操作的扩散、基于 GF(2^4)域和 GF(2^8)域的扩散算法以及基于 GF(17)域和 GF(257)域的扩散算法。将置乱和扩散算法按图 3-1 组合起来，可以实现常规的混沌数字图像密码系统，第 4 章将研究混沌数字图像密码系统的性能评价方法。