第五章 时空域下的信息隐藏

时空域是对应于变换域而言的。在第三章中,我们已经明确了变换域的概念。我们试图给时空域下一个定义,但事实上这种定义的意义不大。简而言之,不对信号作任何频率变换而得到的信号域就是时空域。按照物理上相对论的理解,时间与空间是对立统一的,但在这里,我们并不严格地区分它们。一个放在我们面前的图像,既可以认为是一个时域信号(注意:数字图像是要经过时间抽样而得到的),当然也可以认为是一个平面空间。这种认识的差异仅仅是分析的角度不同罢了。在接下来的内容中,我们是不区别这两个概念的。

对于图像载体,其信号空间也就是像素值的取值空间。单纯的谈像素实际上是不科学的,因为对同一图像点,有不同的坐标去描述它。所以,我们选择了 RGB 空间下的像素(RGB 像素)与 YCbCr 空间下的像素(YUV 像素)作为分析对象,研究了基于 RGB 颜色空间和图像亮度空间的空域隐藏。此外,由于二值图像的特殊性,我们专门实现了二值图像作为载体的隐藏实验。

5.1 基于图像 RGB 颜色空间的信息隐藏

在第一章中我们已经介绍了 RGB 颜色空间以及图像的 RGB 像素的构成。我们在后面谈到的像素都是指 RGB 像素:p(r, g, b)。在这样一个像素中,r, g, b 三个颜色分量都是取值(0, 1)的,为了更好地与“位”这一概念相联系,本节中,我们使用的图像存储方式不再是双精度浮点型(double),而是采用无符号 8 位整型(uint8)进行操作,二者的对应关系如图 5.1 所示。

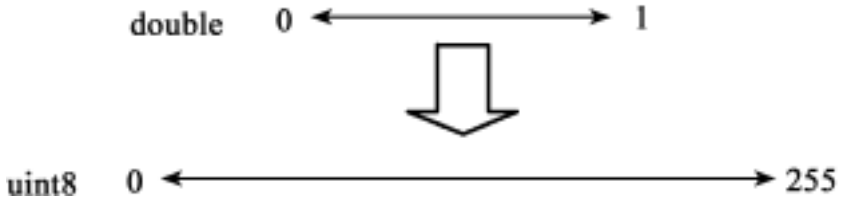


图 5.1 精度对应关系

那么,在 uint8 中修改单位 1,实际上就是对应修改像素值 1/256 = 0.0039。

5.1.1　LSB 与 MSB

LSB(Least Significant Bits) 对应的中文意思是: 最不重要位, 有时也称为最低有效位或简称最低位。MSB(Most Significant Bits) , 是最重要位。那么这里的重要是指对什么重要呢? 如何重要呢? 我们来看下面一个实验。

取 lenna 图像为分析对象。

操作一, 将其各个像素点各个分量的 LSB 清 0, 操作如下:

```
>> data = imread( 'c: \lenna. jpg' ) ;
>> data = bitand( x, 254) ; % 与 11111110 进行与运算
>> subplot( 121) , imshow( data) , title( '清 LSB 后的结果' ) ;
```

操作二, 将其各个像素点各个分量的 MSB 清 0, 操作如下:

```
>> data = imread( 'c: \lenna. jpg' ) ;
>> data = bitand( x, 127) ; % 与 01111111 进行与运算
>> subplot( 122) , imshow( data) , title( '清 MSB 后的结果' ) ;
```

两个操作后的结果如图 5. 2 所示。显然, 图 5. 2(b) 是修改了 MSB 的结果, 图像色彩已经完全被破坏了, 而图 5. 2(a) 依然清晰可见。

结合 RGB 颜色模型我们再来分析一下, 前面我们说过, 在 uint8 格式下修改 1 个单位对应的像素值是 0. 0039。对于操作一, r, g, b 三个分量最大的可能是同时减小 0.0039, 在 RGB 立方体中对应的色彩偏移是: $\sqrt{0.0039^2+0.0039^2+0.0039^2}=0.0068$。而对于操作二, r, g, b 三个分量最大的可能是同时减小 $128\times 0.0039=0.4992$, 在 RGB 立方体中对应的色彩偏移是: $\sqrt{0.4992^2+0.4992^2+0.4992^2}=0.8646$。可以看到, 修改 MSB 对图像的影响可以是修改 LSB 的 127 倍, 同时 0. 8646 的偏移基本上可以使一个像素色彩转变为它的互补色, 如图 5. 3 所示。

图 5. 2　修改 LSB 与 MSB 的结果

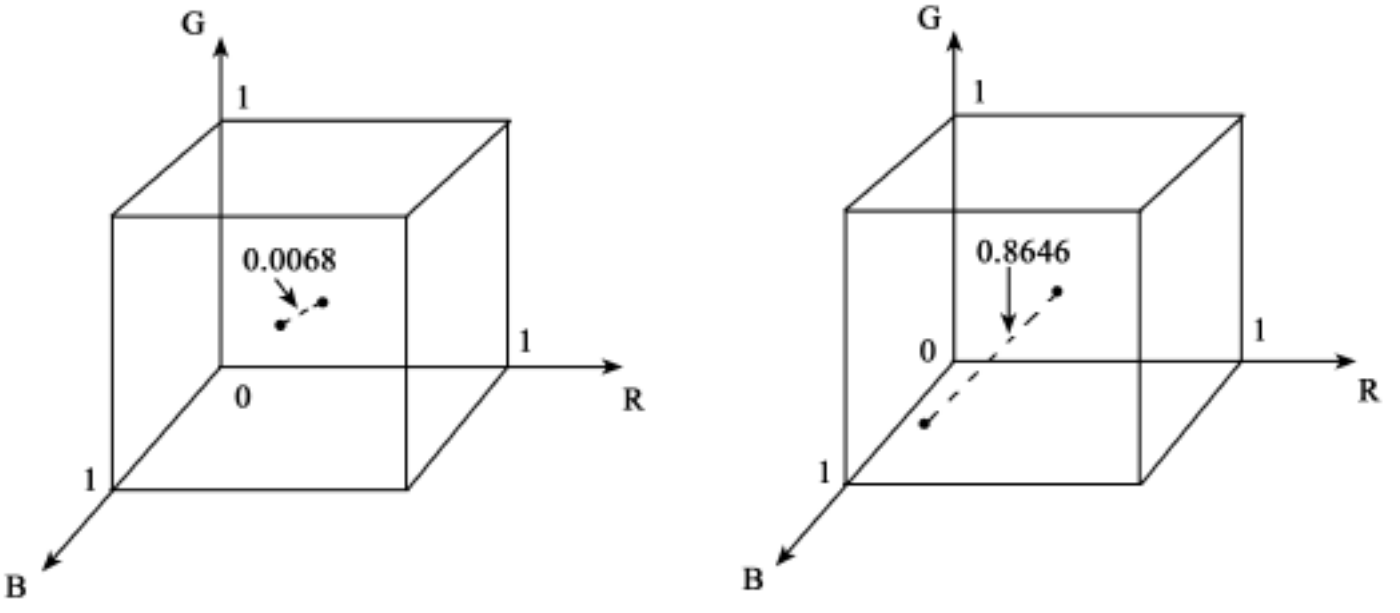


图 5.3　修改 LSB 与 MSB 的影响

从上面的实验可以看到, LSB 所蕴涵的信号对于图像整体来说, 的确是最低和有效的。我们将这种信号在一定意义上理解为是一种冗余。这种冗余, 为我们有效地进行信息隐藏提供了宿空间。

5.1.2　在 LSB 上的信息隐秘

(1) 顺序选取图像载体像素, 将消息隐秘于 LSB

这一节我们将利用最不重要位实现图像载体下的信息隐秘。隐秘算法核心是将我们选取的像素点的最不重要位依次替换成秘密信息, 以达到信息隐秘的目的。嵌入过程包括选择一个图像载体像素点的子集$\{j_1, \cdots, j_{l(m)}\}$, 然后在子集上执行替换操作像素 $c_{j_i} \backslash m_i$, 即把 c_{j_i}的 LSB 与秘密信息 m_i 进行交换(m_i 可以是 1 或 0) 。一个替换系统也可以修改载体图像像素点的多个比特, 例如, 在一个载体元素的两个最低比特位隐藏两比特、三比特信息, 可以使得信息嵌入量大大增加但同时将破坏载体图像的质量。在提取过程中, 找出被选择载体图像的像素序列, 将 LSB(最不重要位) 排列起来重构秘密信息, 算法描述如下:

嵌入过程: for (i = 1; i < = 像素序列个数; i ++)

$s_i \leftarrow c_i$

for (i = 1; i < = 秘密消息长度; i ++)

//将选取的像素点的最不重要位依次替换成秘密信息

$s_{j_i} \leftarrow c_{j_i} \backslash m_i$

提取过程: for (i = 1; i < = 秘密消息长度; i ++)

{

$i \backslash j_i$　　//序选取

$m_i \leftarrow LSB(c_{j_i})$

}

接下来我们具体应用这两个算法来实现秘密消息的隐藏与提取。编写函数lsbhide. m完成实验。函数 lsbhide. m 的功能是将一个给定的秘密消息，将其隐藏于灰度图像载体中。

```
% 文件名: lsbhide. m
% 程序员: 李巍
% 编写时间: 2004. 2. 29
% 函数功能: 本函数将完成在 LSB 上的顺序信息隐秘，载体选用灰度 BMP 图
% 输入格式举例: [ ste_cover, len_total] = lsbhide( 'glenna. bmp', 'message. txt',
'cover. bmp')
% 参数说明:
% input 是信息隐蔽载体图像，为灰度 BMP 图
% file 是秘密消息文件
% output 是信息隐秘后生成图像
% ste_cover 是信息隐秘后图像矩阵
% len_total 是秘密消息的长度，即容量
function [ ste_cover, len_total] = lsbhide( input, file, output)
% 读入图像矩阵
cover = imread( input) ;
ste_cover = cover;
ste_cover = double( ste_cover) ;
% 将文本文件转换为二进制序列
f_id = fopen( file, 'r') ;
[ msg, len_total] = fread( f_id, 'ubit1') ;
% 判断嵌入消息量是否过大
[ m, n] = size( ste_cover) ;
if len_total > m* n
        error( '嵌入消息量过大，请更换图像') ;
end
% p 作为消息嵌入位数计数器
p = 1;
for f2 = 1: n
    for f1 = 1: m
        ste_cover( f1, f2) = ste_cover( f1, f2) -mod( ste_cover( f1, f2) , 2) + msg( p, 1) ;
        if p == len_total
```

```
                break;
            end
            p = p + 1;
        end
        if p  == len_total
            break;
        end
    end
end
ste_cover = uint8( ste_cover) ;
% 生成信息隐秘后图像
imwrite( ste_cover, output) ;
% 显示实验结果
subplot( 1, 2, 1) ; imshow( cover) ;  title( '原始图像 ') ;
subplot( 1, 2, 2) ; imshow( output) ; title( '隐藏信息的图像') ;
```

利用 lsbhide. m 函数，我们得到以下实验结果，如图 5. 4 所示。很明显，通过肉眼我们是看不出两幅图有什么差别的，也就是说此隐秘算法的不可见性还是比较好的。我们编写了一个函数 compare. m，使读者能够直观看出隐藏的位置以及两幅图像的区别。

图 5. 4　LSB 空域信息隐秘后图像与原始图像对比

```
% 文件名: compare. m
% 程序员: 李巍
% 编写时间: 2004. 3. 3
% 函数功能: 本函数完成显示隐秘前后两幅图像的区别
% 输入格式举例: F = compare( 'lenna. bmp', 'cover. bmp')
% 参数说明:
```

```
% original 是原始载体图像
% hided 是隐秘后的图像
% F 是差值矩阵
function F = compare( original, hided)
% 读取原始载体图像矩阵
W = imread( original) ;
W = double( W) /255;
% 读取隐秘后图像矩阵
E = imread( hided) ;
E = double( E) /255;
% 将两图像矩阵相减, 显示效果
F = E-W; % 注意, MATLAB 中矩阵相减只支持 double 型
imshow( mat2gray( F) )
```

利用函数 compare. m, 我们得到以下的实验结果, 如图 5. 5 所示。

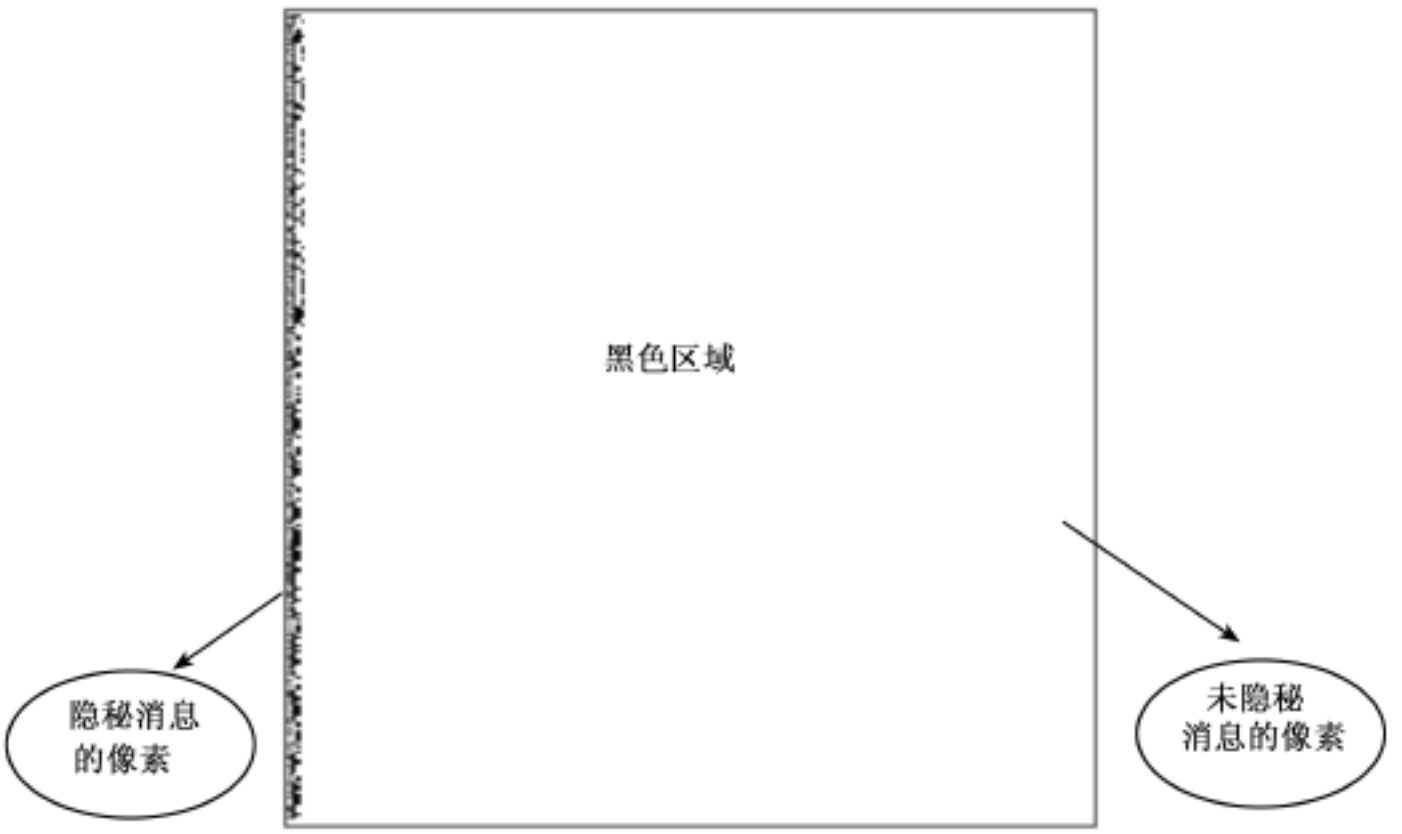
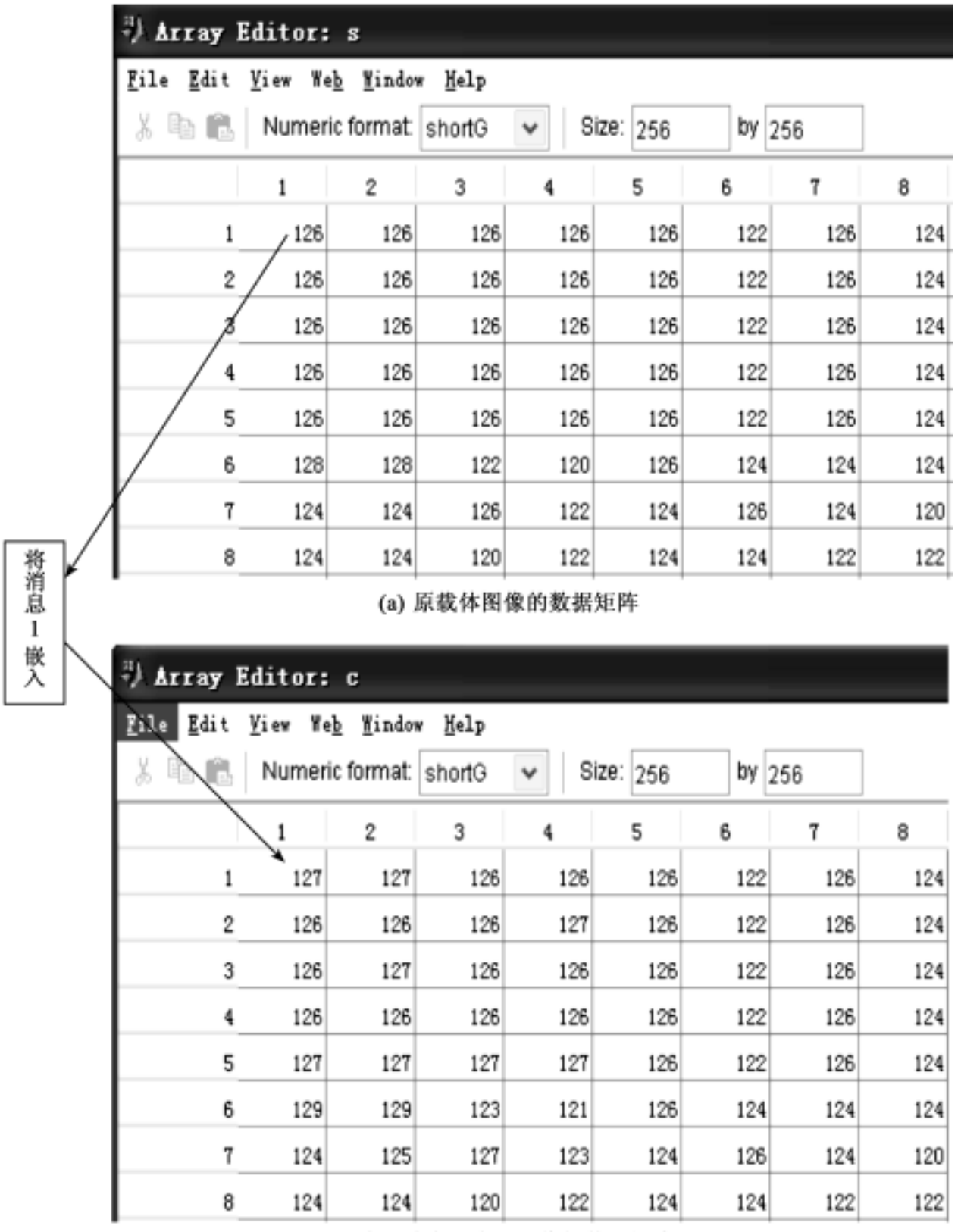


图 5.5　LSB 空域信息隐秘位置

图 5. 5 是经过反色处理的, 左侧的黑色部分是两幅图不一样之处, 也就是隐秘消息的地方。因为是顺序隐秘, 所以我们对于图像的变动都在某一部分区域, 虽然比较容易实现, 但很容易被检测出; 又因为在一幅图像中有两部分的统计特性不一致, 故会导致严重的安全问题。为了解决这个问题, 我们将在下面一小节, 用随机间隔选取

像素点来取代顺序选取。

我们截取了两幅图的数据矩阵，如图 5.6 所示，也可以比较出隐秘消息的地方。



	1	2	3	4	5	6	7	8
1	126	126	126	126	126	122	126	124
2	126	126	126	126	126	122	126	124
3	126	126	126	126	126	122	126	124
4	126	126	126	126	126	122	126	124
5	126	126	126	126	126	122	126	124
6	128	128	122	120	126	124	124	124
7	124	124	126	122	124	126	124	120
8	124	124	120	122	124	124	122	122

(a) 原载体图像的数据矩阵

	1	2	3	4	5	6	7	8
1	127	127	126	126	126	122	126	124
2	126	126	126	127	126	122	126	124
3	126	127	126	126	126	122	126	124
4	126	126	126	126	126	122	126	124
5	127	127	127	127	126	122	126	124
6	129	129	123	121	126	124	124	124
7	124	125	127	123	124	126	124	120
8	124	124	120	122	124	124	122	122

(b) 隐秘消息后生成图像的数据矩阵

图 5.6　LSB 空域信息隐秘数据矩阵

下面我们用函数 lsbget.m 来提取嵌入图像的秘密文本消息，函数代码如下：

```
% 文件名: lsbget.m
```

```
% 程序员: 李巍
% 编写时间: 2004. 2. 29
% 函数功能: 本函数将完成提取隐秘于 LSB 上的秘密消息
% 输入格式举例: result = lsbget( 'cover. bmp', 56, 'secret. txt')
% 参数说明:
% output 是信息隐秘后的图像
% len_total 是秘密消息的长度
% goalfile 是提取出的秘密消息文件
% result 是提取的消息
function result = lsbget( output, len_total, goalfile)
ste_cover = imread( output) ;
ste_cover = double( ste_cover) ;
% 判断嵌入消息量是否过大
[ m, n]  = size( ste_cover) ;
frr = fopen( goalfile, 'a') ;
% p 作为消息嵌入位数计数器, 将消息序列写回文本文件
p = 1;
for f2 = 1: n
    for f1 = 1: m
        if bitand( ste_cover( f1, f2) , 1)  == 1
            fwrite( frr, 1, 'bit1') ;
            result( p, 1)  = 1;
        else
            fwrite( frr, 0, 'bit1') ;
            result( p, 1)  = 0;
        end
        if p  == len_total
            break;
        end
        p = p + 1;
    end
        if p  == len_total
            break;
        end
end
fclose( frr) ;
```

利用函数 lsbget. m, 我们成功地提取出秘密文本文件, 结果如图 5. 7 所示。

图 5. 7　LSB 空域信息隐秘结果

(2) 随机选取像素点, 将消息隐秘于 LSB

如果顺序选取像素点进行信息隐秘, 势必会造成图像各部分统计特征的不一致, 而导致了严重的安全问题。载体的第一部分与第二部分, 也就是修改的部分和没有修改的部分, 具有不同的统计特性, 增大了攻击者对秘密通信怀疑的可能性。为了解决这个问题, 可以随机间隔选取像素序列。此嵌入与提取算法与前面一小节所述的顺序隐秘的算法基本相同, 只是在选取像素序列时不再顺序而是随机间隔的选取。我们编写函数 randlsbhide. m 来实现嵌入算法(需调用随机间隔函数 randinterval. m, 此函数在第二章中有详细介绍) , 函数代码如下:

```
% 文件名: randlsbhide. m
% 程序员: 李巍
% 编写时间: 2004. 3. 2
% 函数功能: 本函数将完成随机选择 LSB 的信息隐秘, 载体选用灰度 BMP 图
% 输入格式举例: [ste_cover, len_total] = randlsbhide('glenna. bmp', 'message. txt', 'scover. bmp', 2001)
```

```
% 参数说明:
% input 是信息隐蔽载体图像
% file 是秘密消息文件
% output 是信息隐秘后的生成图像
% key 是随机间隔函数的密钥
function [ ste_cover, len_total] = randlsbhide( input, file, output, key)
% 读入图像矩阵
cover = imread( input) ;
ste_cover = cover;
ste_cover = double( ste_cover) ;
% 将文本文件转换为二进制序列
f_id = fopen( file, 'r') ;
[ msg, len_total] = fread( f_id, 'ubit1') ;
% 判断嵌入消息量是否过大
[ m, n] = size( ste_cover) ;
if len_total > m* n
    error( '嵌入消息量过大, 请更换图像') ;
    end
    % p 作为消息嵌入位数计数器
    p = 1;
    % 调用随机间隔函数选取像素点
    [ row, col] = randinterval( ste_cover, len_total, key) ;
    % 在 LSB 隐秘消息
    for i = 1: len_total
        ste_cover( row( i) , col( i) ) = ste_cover( row( i) , col( i) ) -mod( ste_cover( row
( i) , col( i) ) , 2) + msg( p, 1) ;
        if p == len_total
            break;
        end
        p = p + 1;
    end
    ste_cover = uint8( ste_cover) ;
    imwrite( ste_cover, output) ;
    % 显示实验结果
    subplot( 1, 2, 1) ; imshow( cover) ; title( '原始图像') ;
    subplot( 1, 2, 2) ; imshow( output) ; title( '隐藏信息的图像') ;
```

利用函数 randlsbhide. m, 我们可以得到实验结果如图 5. 8 所示。

图 5. 8　随机控制下的 LSB 空域信息隐秘后图像与原始图像对比

同样, 用肉眼我们是看不出两幅图像之间的差别的, 也就是说其算法的不可感知性是相当出色的。为了使大家能够直观地看出隐藏的位置以及两幅图像的区别, 再次调用函数 compare. m, 得到如图 5. 9 所示的结果。

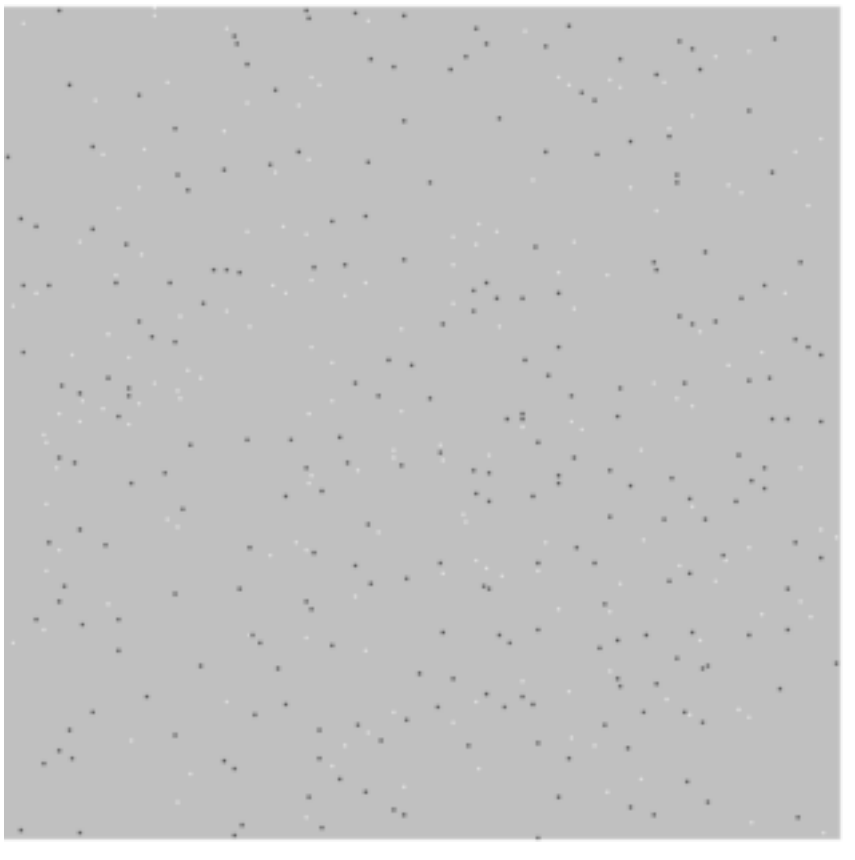


图 5. 9　随机控制下的 LSB 空域信息隐秘位置

图中黑白点都是成功地将消息随机地嵌入到载体图像中的, 而且, 载体图像的统计特征不会因为信息隐秘而不一致。我们截取了两幅图的数据矩阵, 如图 5. 10 所示。

Array Editor: s

File Edit View Web Window Help

Numeric format: shortG　Size: 256 by 256

	1	2	3	4	5	6	7	8
1	126	126	126	126	126	122	126	124
2	126	126	126	126	125	122	126	124
3	126	126	126	126	126	122	126	124
4	126	126	126	126	126	122	126	124
5	126	126	126	126	126	122	126	124
6	128	128	122	120	126	124	124	124
7	124	124	126	122	124	126	124	120
8	124	124	120	122	124	124	122	122

(a)原载体图像的数据矩阵

Array Editor: H

File Edit View Web Window Help

Numeric format: shortG　Size: 256 by 256

	1	2	3	4	5	6	7	8
1	127	127	127	126	127	122	127	125
2	127	127	127	126	127	122	127	125
3	127	127	127	126	127	122	127	125
4	127	127	127	126	127	122	127	125
5	127	127	127	126	127	122	127	125
6	128	128	123	121	126	124	124	125
7	125	125	127	123	125	126	124	121
8	124	124	121	122	124	124	122	123

(b)隐秘消息后生成图像的数据矩阵

图 5.10　随机控制下的 LSB 空域信息隐秘图像的数据矩阵

隐藏与提取是互为逆运算的。我们编写函数 randlsbget. m 提取秘密消息，函数代码如下：

```
% 文件名: randlsbget. m
% 程序员: 李巍
% 编写时间: 2004. 2. 29
% 函数功能: 本函数将完成提取隐秘于 LSB 上的秘密消息
% 输入格式举例: result = randlsbget( 'cover. jpg', 56, 'secret. txt', 2001)
% 参数说明:
% output 是信息隐秘后的图像
% len_total 是秘密消息的长度
% goalfile 是提取出的秘密消息文件
% key 是随机间隔函数的密钥
% result 是提取的消息
function result = randlsbget( output, len_total, goalfile, key)
ste_cover = imread( output) ;
ste_cover = double( ste_cover) ;
% 判断嵌入消息量是否过大
[ m, n] = size( ste_cover) ;
frr = fopen( goalfile, 'a') ;
% p 作为消息嵌入位数计数器, 将消息序列写回文本文件
p = 1;
% 调用随机间隔函数选取像素点
[ row, col] = randinterval( ste_cover, len_total, key) ;
for i = 1: len_total
    if bitand( ste_cover( row( i) , col( i) ) , 1) == 1
        fwrite( frr, 1, 'bit1') ;
        result( p, 1) = 1;
    else
        fwrite( frr, 0, 'bit1') ;
        result( p, 1) = 0;
    end
    if p == len_total
        break;
    end
    p = p + 1;
end
fclose( frr) ;
```

利用函数 randlsbget. m, 我们成功地提取了秘密消息, 结果如图 5. 11 所示。

图 5.11　随机控制下的 LSB 空域信息隐秘结果

5.1.3　在 MSB 上的信息隐秘

前面我们提到一种利用图像最不重要位(LSB)的信息隐藏的方法,它的原理是在用字节表示的图像中,改变字节的最低位,该最低位的变化是人眼不易察觉的,又叫做空间上的冗余,把信息隐藏在这里是比较理想的。这种方法的优点是算法简单、容易实现,而且隐藏的信息不易被肉眼发现,但这也恰恰是对它不利的地方。如果把一幅图像的像素数据的最低位去掉并不会影响该图像的视觉效果,那么,若一幅图像的像素的最低有效位在图像的变换和改变中丢失,隐藏者自己也难发现。

任智斌、隋永新等提出了一种在图像中利用像素数据的最高位即最大意义位实现信息隐藏的想法,即以图像为载体的最大意义位(MSB)信息隐藏技术。这样就会进一步提高信息隐藏的安全性与鲁棒性。当然,这绝不是简单的 LSB 算法的复制,因为随意修改图像的 MSB 对图像是一种极大的破坏。

(1) MSB 空域隐藏算法的原理

要想把信息隐藏在载体中,就必须找出载体中存在的冗余空间,我们对一幅图像的颜色采取量化的方法来创造这种冗余空间。例如,把图像量化到 128 种颜色,然后把图像按照 256 色的格式存储。在 256 色存储的图像中,有一个 256 色的调色板,对于一幅索引图像来说,每个像素都用一个 8 位的索引号来表示,因 8 位二进制对应 256 种颜色,而实际上只有 128 种颜色,所以,只用 7 位二进制表示就足够了。把 7 位放在一个像素的索引号的低 7 位,最高的一位便是我们需要的冗余空间,可以用来

隐藏信息。这时我们要注意的问题是调色板的配置,要把调色板按照以 128 为周期的循环来安排,也就是说,调色板中的第 0 号颜色要与第 128 号颜色对应,第 1 号颜色要与第 129 号颜色对应,依此类推。为了方便大家理解,我们举一个例子: A 点的像素值是 00000001,它是指向颜色表的索引号,对应第 1 号颜色,B 点的像素值是 10000001,对应第 129 号颜色,由于调色板是以 128 为周期的循环,这两种颜色是一样的,由此可见,图像像素值的最高位的变化不会影响该点的颜色值,我们可以利用该最高位来隐藏信息。

(2) MSB 算法的实现过程

在具体实现过程中需要考虑以下几个问题:

第一,什么样的图像适宜于这种算法。要达到隐藏的目的,我们需要的是所隐藏的信息不被观察者察觉,但是对于 MSB 方法,由于它要求将图像量化到 128 色,这样,有很多图像就不合乎要求了。因为对于大部分图像来说,它们的颜色远远超过 128 色,特别是那些鲜艳的图像(如风景画等),如果强行把它们改为 128 色,图像将产生严重的失真,因此,我们只能选择那些颜色简单的图像,这一点是该方法的一个很大的缺点,我们也只将这种方法提出来供大家讨论和改进。

第二,选择什么样的量化方法。量化方法的好坏将直接影响隐藏的效果,要求量化后的图像与原来的图像越接近越好。现有的量化方法也很多,要尽量选择较好的量化方法。

第三,量化后的处理。再怎么好的量化方法,也是对图像有损坏的,量化过程是用与之比较接近的颜色来表示该颜色,这样做势必会产生量化的误差,甚至会出现分层现象。我们来看一看效果:

用下面的代码在 MATLAB 中显示一幅真彩色图像:

```
>> RGB = imread( 'flowers.tif');
>> image( RGB);
```

得到如图 5.12 所示的图像,再用以下代码将这幅真彩色图像转化为 128 色的索引图像,如图 5.13 所示。

```
>> [X, MAP] = RGB2IND( RGB, 128);
>> image( X);
```

可以明显地看到,这样的转化对图像的破坏是很大的。为了解决这个问题,对量化后的图像进行处理、修正误差是非常重要的。一般地说,在对连续色调图像进行颜色量化处理时,由于存在量化误差,图像的视觉质量会降低,甚至会出现像图 5.13 那样的图像,在颜色均匀渐变的区域出现所谓的伪轮廓,因此,必须引进 Gentile 在颜色量化算法中的半色调技术。所谓半色调技术是指在连续灰度图像二值量化输出时为了补偿量化误差对图像质量的负效应所采用的一种图像处理方法,而目前最为常用的半色调技术为误差扩散技术。误差扩散技术的核心是一个对量化误差进行频谱整形的数字滤波器。Floyd 和 Steinberg 所设计的滤波器在量化误差为白噪声时的效果

最为理想。

图 5.12　一幅真彩色图像

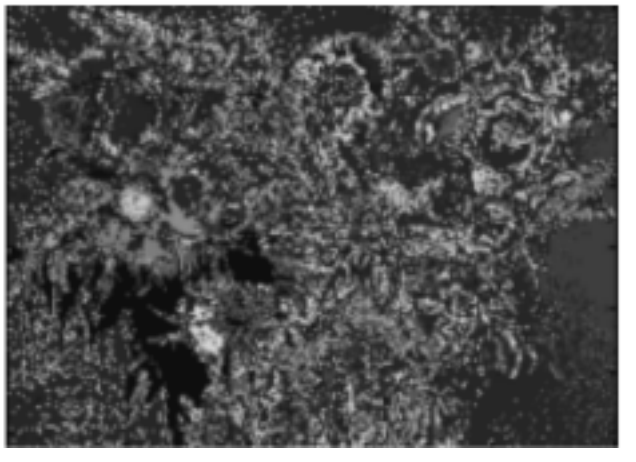


图 5.13　颜色量化到 128 色后的图像

误差扩散算法的原理最早由 Floyd 和 Steinberg 在 1975 年首次提出: 把由图像量化对图像单元的影响(即误差) 传递给与该点相邻的那些像素点, 这样, 图像在显示时的显示误差由于其相邻的像素也被影响而得到补偿, 不易被发现。经误差扩散处理所得半色调图像的局域平均值等于原连续色调图像。图 5.14 是该算法的原理图。

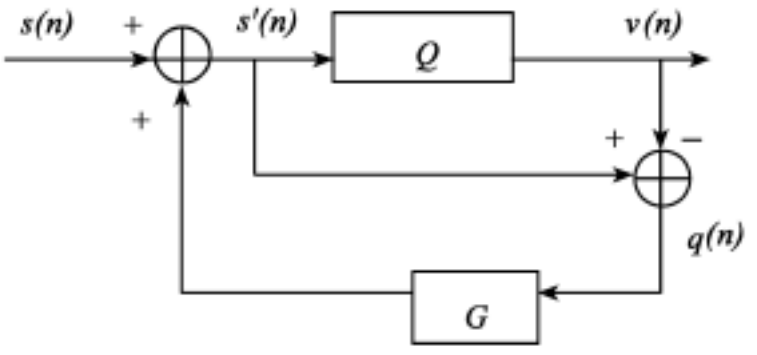


图 5.14　误差扩散算法的原理框图

图 5.14 中的 s(n) 为连续色调图像, Q 为量化器, G 为一个误差滤波器, 其脉冲响应为 q(n) , FS 滤波器的结构如图 5.15 所示。

图 5.15 中 n 为当前处理的图像单元, 0 为已处理的单元, $g_{1,0}$, $g_{1,1}$, $g_{-1,1}$, $g_{0,1}$ 为未处理单元, 滤波器的脉冲响应 $g_{1,0} = 7/16$, $g_{1,1} = 1/16$, $g_{0,1} = 5/16$, $g_{-1,1} = 3/16$, 其滤波过程是将当前正在处理像素的量化误差以上述权重传递给未处理的图像, 而整个误差扩散算法可以等价为一个二维数字滤波器, 分析图 5.14:

$$s'(n) = s(n) + \sum g(n) * q(n) \tag{5.1}$$

$$y(n) = Q[s'(n)] \tag{5.2}$$

$$q(n) = s'(n) - y(n) \tag{5.3}$$

q(n) 即为量化误差, 将式(5.1) 代入式(5.3) 可得:

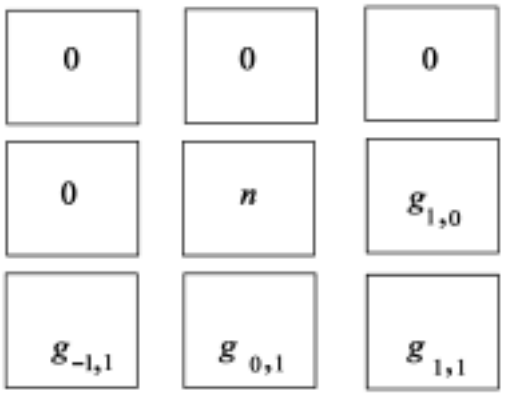


图 5.15　FS 滤波器结构

$$q(n) = s(n) - y(n) + \sum g(n) * q(n)$$
$$= e(n) + \sum g(n) * q(n) \tag{5.4}$$

其中 $e(n) = s(n) - y(n)$ 为显示误差，对式(5.4)做 Fourier 变换可得：

$$E(\omega) = [1 - G(\omega)] * Q(\omega) \tag{5.5}$$

式(5.5)中 $E(\omega)$，$G(\omega)$，$Q(\omega)$ 分别为 $e(n)$，$g(n)$，$q(n)$ 的 Fourier 变换式，$\omega = (u, v)$ 表示二维的空间频率，$D(\omega) = 1 - G(\omega)$ 为误差扩散的传递函数。根据要求，我们把 $G(\omega)$ 设计成低通滤波器，那么 $Q(\omega)$ 为高通滤波器，这时，当量化误差为白噪声时，经过此滤波器后在低频受到抑制，在高频获得增益，从而使显示误差的能量转移到高频谱段，这样的显示误差也称为图像的蓝色噪声，因而将显示误差频谱分布进行整形的过程称为误差蓝化。经过这样的处理，人眼往往无法察觉图像高频段的变化，使得这样的隐藏算法成为可能。

(3) MSB 算法用于隐藏和提取

同 LSB 的隐藏方法一样，传递信息的双方必须就隐藏信息的位置达成一致，在 MSB 算法中，我们当然是将秘密信息隐藏于 8 位像素的最高位，因此，双方要达成一致的是将信息藏入了哪些像素，这可以通过由共同的密钥来产生随机序列进行控制。具体对于像素位修改的方法是：首先将像素的冗余位清零，再与秘密信息的一位进行模 2 加，然后选取下一个像素和下一位信息。就隐秘算法本身来讲，仍然是在像素的 RGB 空间中直接进行的。

本小节简单地介绍了一种新的空间域信息隐藏的方法，这种方法的提出是基于考虑到像素的最高位对视觉产生的影响较大，如果这个最大意义位遭到篡改，是很容易被信息的传递者发现的，这样就保证了隐藏信息的安全，同时，与最低有效位隐藏的方法比较而言，最高有效位不是那么容易丢失和遭到修改的。这种算法的核心是对图像色彩的处理和滤波器的构造。当然，这种方法在图像的选取、操作的复杂性上是不如 LSB 方法的，至于如何去改进它，还有待大家共同思考。

5.2　二值图像中的信息隐藏

所谓二值图像，就是将一个多灰度级的输入图像经过处理后变成只有两个灰度（RGB 像素值为(0,0,0)与(1,1,1)）的图像。二值图像在日常生活中运用得是比较广泛的，如传真图像等。本节中，我们将具体实现 Zhao 和 Koch 提出的一种方案。

5.2.1　算法描述

Zhao 和 Koch 提出了一个信息隐藏方案，他们使用一个特定图像区域中黑像素的个数来编码秘密信息。把一个二值图像分成矩形图像区域 B_i，分别令 $P_0(B_i)$ 和 $P_1(B_i)$ 为黑白像素在图像块 B_i 中所占的百分比。算法思想为：若某块 $P_1(B_i) > 50\%$，则嵌入一个 1，若 $P_0(B_i) > 50\%$，则嵌入一个 0。在嵌入过程中，为达到希望的像素关系，需要对一些像素的颜色进行调整。调整的区域是那些与邻近像素有相反颜色的像素。在具有鲜明对比性的二值图像中，应该对黑白像素的边界进行修改，所有的这些规则都是为了确保不被引起察觉。

为了使整个系统对传输错误和图像修改具有鲁棒性，必须调整嵌入处理过程。如果在传输过程中一些像素改变了颜色，例如 $P_1(B_i)$ 由 50.6% 下降到 49.5% 时，这种情况就会发生，从而破坏了嵌入信息。因此，要引入两个阈值 $R_1 > 50\%$ 和 $R_0 < 50\%$ 以及一个健壮参数 λ，λ 是传输过程中能改变颜色的像素百分比。发送者在嵌入处理中确保 $P_1(B_i) \in [R_1, R_1 + \lambda]$ 或 $P_0(B_i) \in [R_0 - \lambda, R_0]$。如果为达到目标必须修改太多的像素，就把这块标识成无效，其修正方法为：

$$P_1(B_i) < R_0 - 3\lambda \text{ 或}$$

$$P_1(B_i) > R_1 + 3\lambda$$

然后以比特 i 伪随机地选择另一个图像块。在提取过程中，无效的块被跳过，有效的块根据 $P_1(B_i)$ 进行。

5.2.2　算法中的几个值得注意的问题

(1) 检查可用的图像块

前面的算法中，对于图像块的要求是十分严格的。图像块是否可用，就是看 $P_1(B_i)$ 的大小（$P_0(B_i)$ 与 $P_1(B_i)$ 是否互补的，考虑其中一个就行了）。经过分析可得，$P_1(B_i)$ 有以下几种情况：

$$\begin{cases} P_1(B_i) > R_1 + 3\lambda \quad \text{or} \quad P_1(B_i) < R_0 - 3\lambda & \text{(a)} \\ R_0 - 3\lambda < P_1(B_i) < R_0 \quad \text{or} \quad R_1 < P_1(B_i) < R_1 + 3\lambda & \text{(b)} \\ R_0 - 3\lambda < P_1(B_i) < R_1 & \text{(c)} \\ R_0 < P_1(B_i) < R_1 + 3\lambda & \text{(d)} \\ R_0 - \lambda < P_1(B_i) < R_0 \quad \text{or} \quad R_1 < P_1(B_i) < R_1 + \lambda & \text{(e)} \end{cases}$$

(a)对应的是不可用区域,即当$P_1(B_i)$属于这个范围时,表示对该块不加任何操作而跳过。在隐秘算法中,图像载体某一块的$P_1(B_i)$可能本来就是属于(a),则不进行任何操作。$P_1(B_i)$也可能被调整到这一区域,表示这一块与要嵌入的数据不匹配而舍去。

(b)是最复杂的一种情况。我们不能简单地认定$P_1(B_i)$在这一区域中的图像块是否可用,而应该与秘密信息对应来分析。如果该块试图隐藏的信息是0,而$R_1 < P_1(B_i) < R_1 + 3\lambda$,则相应的$P_0(B_i)$过小,与我们最初定义的"$P_0(B_i) > 50\%$,则嵌入一个0"很不匹配,不能选用这一块用于表示信息0的隐藏;同理,如果该块试图隐藏的信息是1,而$R_0 - 3\lambda < P_1(B_i) < R_0$,则相应的$P_1(B_i)$过小,与我们最初定义的"$P_1(B_i) > 50\%$,则嵌入一个1"很不匹配,也不能选用这一块用于表示信息1的隐藏。在这种情况下,我们称$P_1(B_i)$在(b)的块为难以调整块。但如果我们要嵌入的信息是1,而$R_1 < P_1(B_i) < R_1 + 3\lambda$或者要嵌入的是0而$R_0 - 3\lambda < P_1(B_i) < R_0$,则是毫无影响的,这种情况就被包括在(c)和(d)中了。

(c),(d)分别表示可以嵌入1和可以嵌入0的情况。即$P_1(B_i)$在这些区域的图像块是我们重点操作的块。当然,"可以嵌入"只表明了可能性,具体要嵌入的是0还是1则由秘密信息决定。(b)与之有重叠是值得注意的,在(b)得到正确处理的条件下,我们最终要将$P_1(B_i)$从(c),(d)调整为(e),并严格地约束$P_1(B_i)$的范围,从而提高信息隐藏的鲁棒性。图5.16是$P_1(B_i)$的取值范围与对应的图像块的直观关系图。

(2)对$P_1(B_i)$的调整

图5.16列出了$P_1(B_i)$与相应的图像块怎样操作的对应关系。同时注意到,图中的虚箭头表示$P_1(B_i)$的调整,具体的调整有两个方面:一是将难以调整块改变为不可用块;二是将可用块改变为最终隐藏块。这样调整也有两个目的:一是使隐蔽载体中不再有$R_0 < P_1(B_i) < R_1$的弱鲁棒块($P_1(B_i)$在该区域中我们认为是容易变化的);二是增大不可用块与最终隐藏块之间的区别,便于信息提取的方便。在信息提取中,隐藏后的图像(stego-cover)中只有符合(a)和(e)两种情况的图像块,对于(a)不进行任何操作,对于(e)则提取相应的信息,二者的最小差距也有2λ。

了解了比例调整的意义后,怎样调整就成为了关键。调整的实质性操作就是将一个白点(1,1,1)改成黑点(0,0,0)或相反操作。若要增大$P_1(B_i)$是将若干白点改成黑点,若要减小$P_1(B_i)$则是将若干个黑点改成白点。如果修改的结果是一片白色

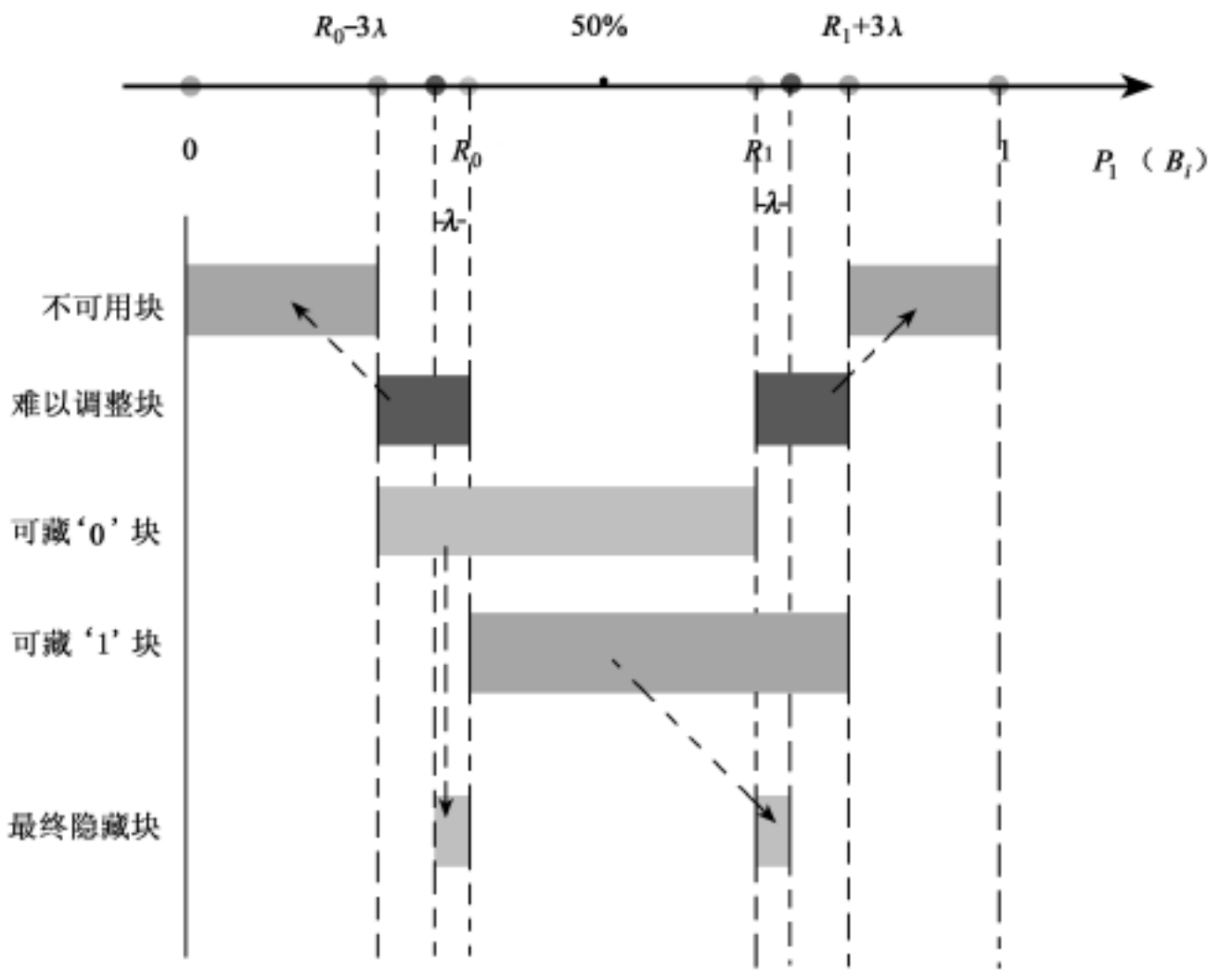


图 5.16　$P_1(B_i)$ 与图像块

中加入了几个黑点或一片黑色中加入了几个白点,都是非常影响隐藏不可见性的。避免这种“万花丛中一点绿”的办法就是寻找边界,继而在边界处修改。白色或黑色区域的边界向外(内)扩大(减小)几个像素的面积则被认为是不可察觉的了。图 5.18是在黑色区域修改两个黑点为白点,在白色区域修改两个白点成黑点的修改状况图。

比较图 5.18 的左右两幅图,显然右边的修改图像与原始图像更为接近一些。当然这样修改并不是最好最合理的,同是边界修改,具体的修改位置不同同样也有优劣之分。细小到像素的层次,这一修改位的选择我们就不作考虑了。但只要是靠上边界的,至少没有明显的对比突出情况发生。图 5.17 是一组实际实验结果的对比,显然,在随意修改的情况下,lenna 小姐很不幸地长上了“麻疹”。

在二值图像中,有四连接和八连接两种像素连接的定义方法。四连接(4-neighbors)定义为:像素(x, y)的上(x - 1, y)、下(x + 1, y)、左(x, y - 1)、右(x, y + 1)构成连接,和别的像素点不构成连接。八连接(8-neighbors)在四连接的基础上增加了对角线上的 4 个像素:(x - 1, y - 1),(x - 1, y + 1),(x + 1, y - 1),(x + 1, y + 1)。采用

图 5.17　二值图像像素的修改(1)

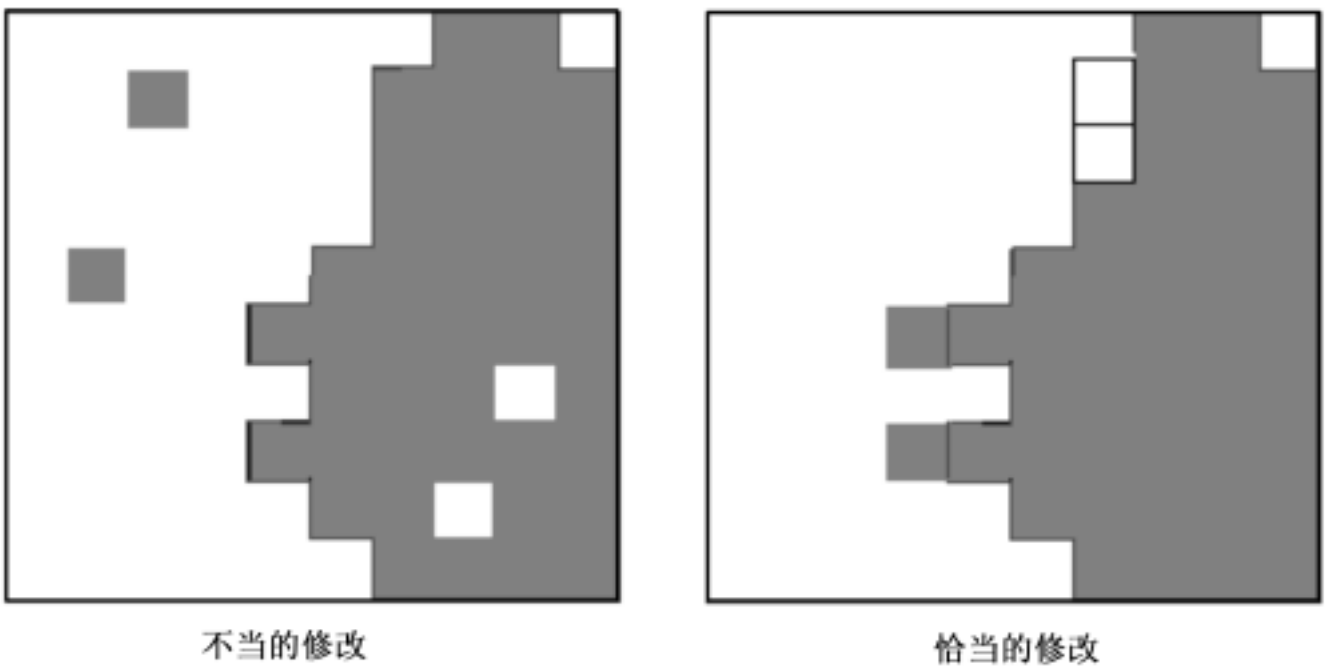


图 5.18　二值图像像素的修改(2)

的连接定义不同,得到的图像边界也不同,图 5.19 表明了这种情况。

显然,四连接与八连接对同一图像得到的边界是不同的,八连接的定义更为广泛。这里,我们采用八连接的定义去判断一个像素点能否被修改,用伪 C 代码描述如下:

```
if ( ( x - 1, y)  ==  !  ( x, y)  ‖  ( x + 1, y)  ==  !  ( x, y)  ‖
     ( x, y - 1)  ==  !  ( x, y)  ‖  ( x, y + 1)  ==  !  ( x, y)  ‖
     ( x - 1, y - 1)  ==  !  ( x, y)  ‖  ( x - 1, y + 1)  ==  !  ( x, y)  ‖
     ( x + 1, y - 1)  ==  !  ( x, y)  ‖  ( x + 1, y + 1)  ==  !  ( x, y) )
  { ( x, y)  = !  ( x, y) ; }
else
   select next pixel;
```

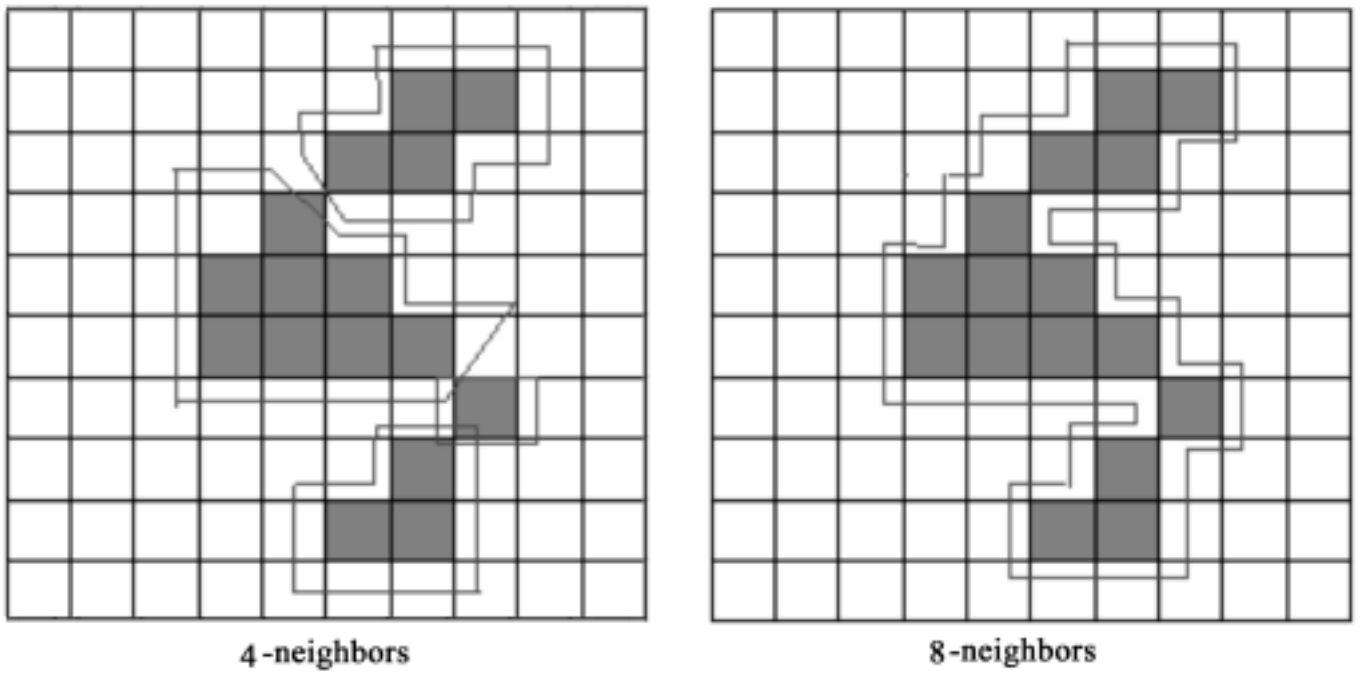


图 5.19　四连接和八连接

按照上述算法进行像素修改，容易出现边界扩散现象。所谓边界扩散现象是说：A是 B 的边界点，A与 B 的颜色不同。C 的八连接点均没有与 C 颜色不同的点存在，那么 B 是可以修改的，C 是不能修改的。但如果 B 恰恰也是 C 的边界点，由于 B 的修改将导致 C 成为可修改点，这样会使原始图像的边界扩大。为了防止 C 可被修改，我们在修改 B 时应将其标记，从而区分原始边界与修改后形成的边界。图 5.20 是边界扩散与无边界扩散对图像不同的影响。

图 5.20　边界扩散

(3)需要考察的参数

在本算法中，我们有三个人为引进的参数：R_0、R_1 和 λ。我们来简单分析一下它们的作用。前面已经提到过，在最终的隐蔽载体中，不存在 $R_0 < P_1(B_i) < R_1$ 的图像块以及 $R_0 - 3\lambda < P_1(B_i) < R_0 - \lambda$ 和 $R_1 + \lambda < P_1(B_i) < R_1 + 3\lambda$ 的图像块。前者的作用是提高隐藏的鲁棒性，避免像素值的轻易变化而导致隐秘信息的丢失；后者是为了在信息隐藏块与不可用块之间设立一个缓冲带，避免在检测的时候将这两类图像块

混淆。应该说,这三个参数的选取都是有讲究的,是值得我们去考察的。我们应该在算法实现中将其作为入口参数参与运算,并且在一定意义上,这三个参数也起着密钥的作用。

5.2.3 算法实现

下面我们就来具体实现 Zhao 和 Koch 的这一算法。在具体实现中,我们是这样解决以下几个问题的:

(1)块的大小与随机选块

为了方便起见,我们就将图像块的大小定义为 10× 10。输入参数 R_0, R_1, λ和输出参数 $P_1(B_i)$ 等都可以直接体现为[0,100]的整数,简化编程难度。

随机选块的方法仍然使用第二章介绍的哈希置换法(用 hashreplacement. m 实现)。不同的是, hashreplacement. m 原本是随机选择像素点的,现在用于随机选块需要对返回参数作一定的修改。我们希望返回的参数是各个块的块首地址。

①在输入时,我们应该以块为单位将图像尺寸输入 hashreplacement. m,而不是将以像素为单位的图像尺寸输入。

②对于返回的行列标,我们应该作如下处理:

```
if ( return value ! =1)
    return value = ( return value - 1)× 10
+1;
```

以一个 256× 256 的图像为例,在随机选块时,我们应该输入的尺寸是 25× 25。假设返回的随机行列标为: (3, 1) , (12, 15) , (2, 9) , (1, 1) , …, (25, 3) ,对应于相应的图像块块首地址则是: (21, 1) , (111, 141) , (11, 81) , (1, 1) , …, (241, 21) 。

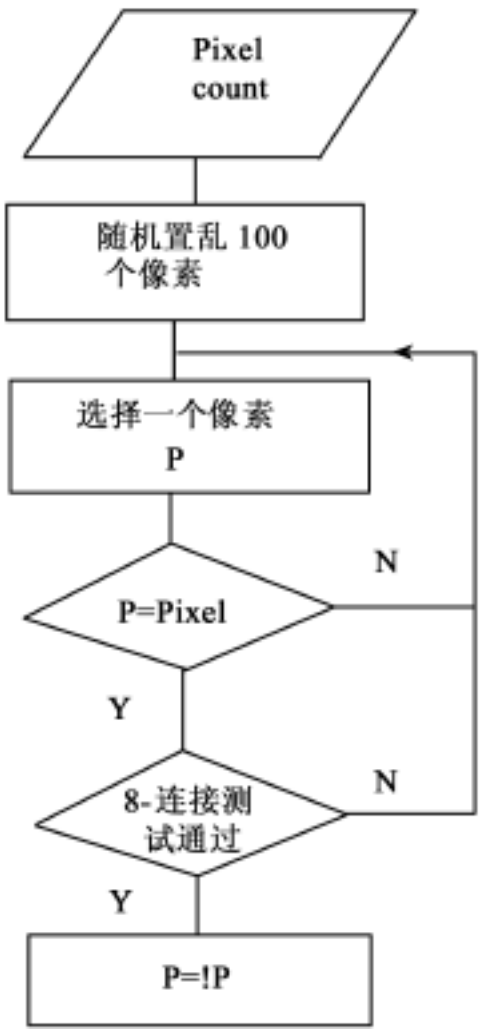


图 5.21 修改像素值的流程图

(2)修改像素值的流程

前面我们已经知道了哪些像素才能被修改,下面是具体的修改流程,如图 5. 21 所示。入口参数为: (pixel, count) 。pixel 为要修改的像素, count 为要修改的数量。在这个流程中,我们仍然用 hashreplacement. m 在 10× 10 的范围内随机置乱 100 个点。然后根据置乱后的顺序找到第一个与所输入的像素值相同的像素,进而判断它的八连接有没有与之相反的像素,如果有则表明这一点可以是边界,可以修改;否则继续往后寻找。同时为了算法效率的提高,我们不必要每一次都

去随机置乱 100 个点,仅进行一次这样的操作然后将置乱的结果作为第三个入口参数输入即可。

值得注意的是,在 10× 10 范围内的所有点并不是都能适用前面的八连接检测算法的。真正能够严格适用的只是中间的 8× 8 个点,所以,我们真正需要随机置乱的是 64 个点而不是 100 个点,这仅仅是为了编写程序的简单,如图 5.22 所示。

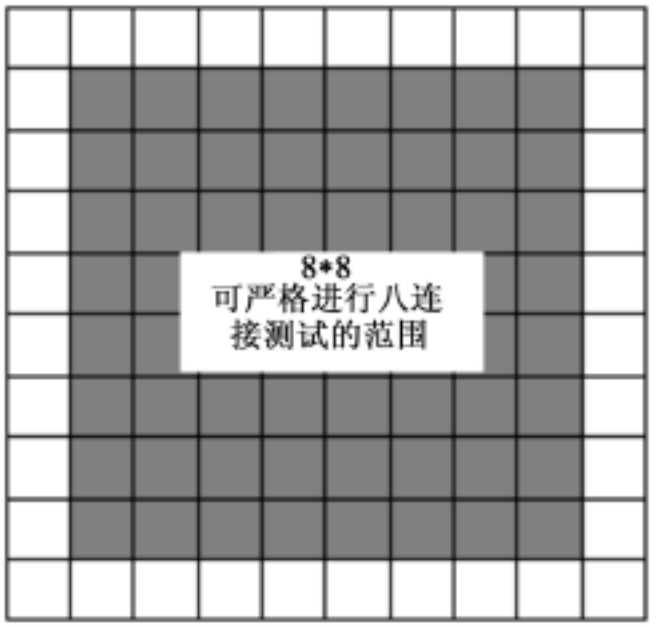


图 5.22　能严格进行八连接测试的区域

那么,结合前面的块首地址,我们可以在每一块中任意确定一个像素。例如,块首地址为(21,41),在这一块中的 8× 8 区域随机选择(1,1),(3,5),(8,8)三个点,其实际的地址为:(22,42),(24,46)和(29,49)。地址转换的方法为:

实际地址 = 块首地址 + 8× 8 区域内的随机地址

(3)修改像素的数量

在前面的流程中,我们引入了一个入口参数 count 为需要修改的像素的数量。结合图 5.16 我们具体分析一下 count 的大小。注意,由于我们的选块策略,所有的比例已经转化为整数了。

如果要嵌入的信息为 1,$P_1(B_i)$ 的可能取值为以下三个区域(见图 5.16):

$$\begin{cases} R_0 < P_1(B_i) < R_1 & (a) \\ R_1 < P_1(B_i) < R_1 + \lambda & (b) \\ R_1 + \lambda < P_1(B_i) < R_1 + 3\lambda & (c) \end{cases}$$

对于(a),需要将 $R_1 - P_1(B_i) + 1$ 个 0 像素改为 1 像素;对于(b),不需要进行任何修改;对于(c)则需要将 $P_1(B_i) - (R_1 + \lambda) + 1$ 个 1 像素改为 0 像素。

如果要嵌入的信息为 0,$P_1(B_i)$ 的可能取值为以下三个区域(见图 5.16):

$$\begin{cases} R_0 < P_1(B_i) < R_1 & (d) \\ R_0 - \lambda < P_1(B_i) < R_0 & (e) \\ R_0 - 3\lambda < P_1(B_i) < R_0 - \lambda & (f) \end{cases}$$

对于(d),需要将 $P_1(B_i) - R_0 + 1$ 个1像素改为0像素;对于(e),不需要进行任何修改;对于(f)则需要将 $(R_0 - \lambda) - P_1(B_i) + 1$ 个0像素改为1像素。

(4)边界不足的情况

在八连接检测的条件下修改像素,可能会出现一种极端情况:结合我们的具体算法实现方法就是遍历了全部64个像素点仍然无法满足修改数量(count)的要求。实验证明,要在64个点中修改10个以上的边界点是很难实现的,原因很简单,就是没有那么多的边界!我们称为边界不足。结合图5.16我们来具体分析如下:

我们对 $P_1(B_i)$ 进行调整,最终使其完全落在 $[0, R_0 - 3\lambda] \cup [R_0 - \lambda, R_0] \cup [R_1, R_1 + \lambda] \cup [R_1 + 3\lambda, 1]$ 区域内,而我们的提取算法描述为:

```
if (P1(Bi) >50% && P1(Bi) < R1 +3λ)
        message = 1;
else if (P1(Bi) <50% && P1(Bi) > R0-3λ)
        message = 0;
```

也就是说,将 $P_1(B_i)$ 存在于 $[R_0, R_1]$ 的块(严格意义上是不应该有这样的块的)一并纳入了可提取的范围,这样就在一定意义上削弱了边界不足的错误对结果的影响。如某块 $P_1(B_i) = 47\%$,该块用于隐藏信息1,取 $R_1 = 55\%$,$\lambda = 2\%$,则应该将其 $P_1(B_i)$ 调整为56%左右,其调整比例为9%。假设现在这一块只可调整6%的像素,在上述提取算法中是不影响的,所以,在本节的算法实践中,我们并没有在程序中对这种情况做深入的分析。当然,仍是上面的例子,如果该块只能调整2%的像素,那么就不行了,并且不符合隐藏鲁棒性的要求(我们设立 R_1, R_0 就是为了提高鲁棒性)。实验表明,适当选择参数 R_1, R_0 和 λ 是可以避免这一现象发生的。

严格地讲,如果某一块真的出现了上述情况,我们也应该将其标记为不可用块。大家有兴趣的话可以试着继续完善它,在程序编写时应用到回溯的思想。

(5)边界扩散的防止

前面已经谈到边界扩散的影响。从图5.20可以明显看出,防止边界扩散是很有必要的。

在具体实验中,我们采用的方法是:

```
if 要修改像素 p
        p = ! p +0.01
```

加上0.01就不会使得这一点在其他点的八连接测试中被通过,从而避免了扩散。当然,这样的像素值是不符合RGB模型的,在处理完后,需要将图像的全部像素取整还原。

(6)函数代码

编写函数 binaryhide. m 和 binaryextract. m 分别完成隐藏和提取的实验。binaryhide. m 函数需要调用三个子函数，分别为：computep1bi. m, availabel. m 和 editp1bi. m。其具体功能在函数代码注释中已有给出。

1) 隐藏主函数：binaryhide. m

```
% 文件名: binaryhide. m
% 程序员: 郭迟
% 编写时间: 2004. 3. 5
% 函数功能: 本函数将完成二值图像下的信息隐秘
% 输入格式举例: [ result, count] = binaryhide( 'c: \blenna. jpg', 'c: \secret. txt', 'c: \test. jpg',1983,45,55,3)
% 参数说明:
% cover 为二值载体图像
% msg 为秘密消息
% goalfile 为保存的结果
% key 为隐藏密钥
% R0, R1 和 lumda 为分析参数
% result 为隐藏结果
% count 为隐藏的信息数
% availabler, availablec 为存放隐藏块首地址的行、列标
function [ result, count, availabler, availablec] = binaryhide( cover, msg, goalfile, key, R0, R1, lumda)
% 按位读取秘密信息
frr = fopen( msg, 'r'); % 定义文件指针
[ msg, count] = fread( frr, 'ubit1'); % msg 为消息的位表示形式, count 为消息的 bit 数
fclose( frr) ;
% 读取载体图像信息
images = imread( cover) ;
image = round( double( images) /255) ;
% 确定图像块的首地址
[ m, n] = size( image) ;
m = floor( m/10) ;
n = floor( n /10) ;
temp = zeros( [ m, n] ) ;
[ row, col] = hashreplacement( temp, m* n, m, key, n) ; % 将 m, n 也作为密钥简化输入
```

```
for i = 1: m* n
    if row( i) ～= 1
        row( i) = ( row( i) - 1) * 10 + 1;
    end
    if col( i) ～= 1
        col( i) = ( col( i) - 1) * 10 + 1;
    end
end
% 随机置乱 8* 8 个点
temp = zeros( 8) ;
[ randr, randc] = hashreplacement( temp, 64, key, m, n) ; % 将 m, n 也作为密钥简化输入
% 分析可用的图像块
[ availabler, availablec, image] = available( msg, count, row, col, m, n, image, R1, R0, lumda, randr, randc) ;
% 信息嵌入
for i = 1: count
    p1bi = computep1bi( availabler( i) , availablec( i) , image) ;
    if msg( i, 1) == 1
        if p1bi < R1
            image = editp1bi( availabler( i) , availablec( i) , image, 0, R1-p1bi + 1, randr, randc) ; % 使 p1( Bi) > R1
        elseif p1bi > R1 + lumda
            image = editp1bi( availabler( i) , availablec( i) , image, 1, p1bi-R1-lumda + 1, randr, randc) ; % 使 p1( Bi) < R1 + λ
        else
        end
    end
    if msg( i, 1) == 0
        if p1bi > R0
            image = editp1bi( availabler( i) , availablec( i) , image, 1, p1bi-R0 + 1, randr, randc) ; % 使 p1( Bi) < R0
        elseif p1bi < R0-lumda
            image = editp1bi( availabler( i) , availablec( i) , image, 0, R0-lumdap1bi + 1, randr, randc) ; % 使 p1( Bi) < R1 + λ
        else
```

```
            end
        end
    end
    % 信息写回保存
    image = round( image) ; % 防止边界扩散后的取整复原
    result = image;
    imwrite( result, goalfile) ;
    subplot( 121) , imshow( images) , title( '原始图像') ;
    subplot( 122) , imshow( result) , title( [ '取阈值 R0, R1 为', int2str( R0) , ',',
int2str( R1) , '以及健壮参数 λ为', int2str( lumda) , '下的信息', int2str( count) , 'bits 隐
秘效果' ] ) ;
```

2) **计算可用图像块的函数**: available. m

```
% 分析可用的图像块与秘密信息对应
% msg, count 为秘密消息及其数量
% row, col 存放的是随机选块后的块首地址的行、列地址值
% m* n 为总块数量
% image 为载体图像
% R1, R0, lumda 为参数
% randr, randc 是在 8* 8 范围内随机置乱的行、列标
function [ availabler, availablec, image] = available( msg, count, row, col, m, n,
image, R1, R0, lumda, randr, randc) ;
msgquan = 1;
unable = 0;
difficult = 0;
for blockquan = 1: m* n
    % 计算这一块的 p1( Bi)
    p1bi = computep1bi( row( blockquan) , col( blockquan) , image) ;
    if p1bi >= R1 + 3* lumda ‖ p1bi < = R0 - 3* lumda % 情况( 1)
        row( blockquan) = - 1; % 标记为无用
        col( blockquan) = - 1;
        unable = unable + 1;
        msgquan = msgquan - 1; % 该消息还未找到可以隐藏的块
    % 情况( b)
    elseif msg( msgquan, 1) == 1 && p1bi < = R0
        % 调整 p1( Bi) 变得更小
        image = editp1bi( row( blockquan) , col( blockquan) , image, 1, 3* lumda,
```

```
randr, randc) ;
                row( blockquan)  =  - 1;
                col( blockquan)  =  - 1;
                difficult = difficult + 1;
                msgquan = msgquan -  1; % 该消息还未找到可以隐藏的块
            elseif msg( msgquan, 1)  == 0  && p1bi >  = R1
                % 调整 p1( Bi) 变得更大
                image = editp1bi( row( blockquan) , col( blockquan) , image, 0, 3*  lumda,
randr, randc) ;
                row( blockquan)  =  - 1;
                col( blockquan)  =  - 1;
                difficult = difficult + 1;
                msgquan = msgquan -  1; % 该消息还未找到可以隐藏的块
            else
                row( blockquan)  = row( blockquan) ;
                row( blockquan)  = row( blockquan) ;
            end
            msgquan = msgquan + 1;
            if msgquan  == count + 1; % 消息已经读取完成
                for i = ( blockquan + 1) : m*  n
                    row( i)  =  - 1;
                    col( i)  =  - 1;
                end
            disp( [ '消息长度: ', num2str( msgquan  -  1) , 'bits; 用到的块数: ', num2str
( blockquan) , '; 其中不可用块有: ', num2 str( unable) , '; 另有', num2 str( difficult) , '块
难以调整块已修改为不可用块'] )
            break;
            end
        end
        % 载体分析完但消息还没有读完
        if msgquan <  = count
            disp ( [ '消息长度: ', num2str ( msgquan  -  1) , 'bits; 用到的块数: ',
num2str( blockquan) , '; 其中不可用块有: ', num2str ( unable ) , '; 另有',
num2str( difficult) , '块难以调整块已修改为不可用块'] )
            disp( '请根据以上数据更换载体! ');
            error( '载体太小! ! ');
```

```
end
% 计算可用块的数量
% disp( row)
quan = 0;
for i = 1: m*  n
        if row( i)  ~= - 1
                quan = quan + 1;
        end
end
if quan < count
        error( 可用块数量太小! 请根据以上数据更换载体! );
end
disp( [ 可用图像块为: ', num2str( quan) ] ) ;
% 生成可用的块的行标列标并与消息对应
image = round( image) ; % 防止边界扩散后的取整复原
availabler = zeros( [ 1, quan] ) ;
availablec = zeros( [ 1, quan] ) ;
j = 1;
for i = 1: m*  n
        if row( i)  ~= - 1;
                availabler( j)  = row( i) ;
                availablec( j)  = col( i) ;
                j = j + 1;
        end
end
```

3) 计算每一块 p1(Bi) 的函数: computep1bi. m

```
% 计算 P1( Bi) 的子函数
% headr 为块首行地址
% headc 为块首列地址
function p1bi = computep1bi( headr, headc, image)
p1bi = 0;
for i = 1: 10
        for j = 1: 10
                if image( headr + i - 1, headc + j - 1)  == 1
                        p1bi = p1bi + 1;
                end
```

```
        end
end
```

4) **修改像素的函数**: editp1bi. m

```
% 修改像素的函数
% headr 为块首行地址
% headc 为块首列地址
% image 为图像
% pixel 为要修改的像素
% count 为修改的数量
% randr, randc 是随机置乱后的结果
function image = editp1bi( headr, headc, image, pixel, count, randr, randc)
c = 0;
for i = 1: 64
        if image( headr + randr( i) , headc + randc( i) ) == pixel
            % 八连接检测
            if image ( headr + randr( i) - 1, headc + randc( i) ) == ~pixel ||
                    image( headr + randr( i) +1, headc + randc( i) ) == ~pixel ||
                    image( headr + randr( i) , headc + randc( i) - 1) == ~pixel ||
                    image( headr + randr( i) , headc + randc( i) +1) == ~pixel ||
                    image( headr + randr( i) - 1, headc + randc( i) - 1) == ~pixel ||
                    image( headr + randr( i) - 1, headc + randc( i) +1) == ~pixel ||
                    image( headr + randr( i) +1, headc + randc( i) - 1) == ~pixel ||
                    image( headr + randr( i) +1, headc + randc( i) +1) == ~pixel
                image( headr + randr( i) , headc + randc( i) ) = ~pixel + 0. 01;
                c = c +1;
            end
        end
        if c == count
            return
        end
    end
    % 出现边界不足的情况
    if c ~= count
        disp( 'warning! 参数选择不当, 未能完全按要求修改本块像素, 信息可能无法提取, 建议重做');
    end
```

5) **信息提取函数**: binaryextract. m

% **函数功能**: **本函数将完成亮度空间下的隐秘信息的提取**

% **输入格式举例**: result = binaryextract('c: \test. jpg', 'c: \extract. txt', 1983, 45, 55, 3, 24)

% **参数说明**:

% stegocover **为隐藏有信息的秘密消息**

% goalfile **为信息提取后保存的地址**

% key **为提取密钥**

% R0, R1 **和** lumda **为分析参数**

% count **为要提取的信息数**

% result **为提取的信息**

```
function result = binaryextract( stegocover, goalfile, key, R0, R1, lumda, count)
% 读取隐蔽载体图像信息, 并提取亮度分量。该载体应为 16 位存储方式的图像, 建议使用 png 格式
stegoimage = imread( stegocover) ;
stegoimage = round( double( stegoimage) /255) ;
% 确定图像块的首地址
[ m, n] = size( stegoimage) ;
m = floor( m/10) ;
n = floor( n/10) ;
temp = zeros( [ m, n] ) ;
[ row, col] = hashreplacement( temp, m* n, m, key, n) ; % 将 m, n 也作为密钥简化输入
for i = 1: m* n
    if row( i) ~= 1
        row( i) = ( row( i) - 1) * 10 + 1;
    end
    if col( i) ~= 1
        col( i) = ( col( i) - 1) * 10 + 1;
    end
end
% 准备提取并回写信息
frr = fopen( goalfile, 'a') ; % 定义文件指针
% 按隐藏顺序分析图像块
quan = 1;
result = zeros( [ count 1] ) ;
```

```
for i = 1: m* n
    % 计算这一块的 p1( Bi)
    p1bi = computep1bi( row( i) , col( i) , stegoimage) ;
    if p1bi < R1 +3* lumda && p1bi >50
        fwrite( frr, 1, 'bit1') ; % 回写 1
        result( quan, 1) =1;
        quan = quan + 1;
    elseif p1bi > R0 - 3* lumda && p1bi <50
        fwrite( frr, 0, 'bit1') ; % 回写 0
        result( quan, 1) =0;
        quan = quan + 1;
    else
        quan = quan;
    end
    if quan == count +1
        break;
    end
end
disp( [ '已经正确处理', num2str( quan - 1) , 'bits 的消息'] ) ;
fclose( frr) ;
```

5.2.4 实验分析

我们具体讨论阈值 R_0, R_1 及健壮参数 λ对实验结果的影响。

(1) 健壮参数 λ的选取

实际的健壮参数 λ是和传输过程中可能发生改变的像素的百分比有关的。λ的作用是区分隐藏有信息的图像块与未隐藏有信息的图像块的,区分的尺度正好是 2λ。只要能达到这个目的,λ的取值就是合适的。下面我们仍然以 lenna 的二值图像为例(二值量化阈值取 0.4)分析在 JPEG 压缩下图像像素改变的比例。

我们编写函数 jpgandlumda.m 函数完成分析,函数代码如下:

```
% 文件名: jpgandlumda. m
% 程序员: 郭迟
% 编写时间: 2004. 3. 10
% 函数功能: 本函数将探讨二值图像在 JPEG 条件下像素改变的状况
% 输入格式举例: jpgandlumda( 'c: \blenna. jpg')
% 参数说明:
% test 为二值图像
```

```
function jpgandlumda( test)
image = imread( test) ;
image = round( double( image) /255) ;
[ M, N]  = size( image) ;
quality = 5: 5: 100; % 定义压缩质量比从 5% 到 100%
result = zeros( [ 1  max( size( quality) ) ] ) ;
count = 0;
different = 0;
for q = quality
        count = count + 1;
        imwrite( image, 'temp. jpg', 'jpg', 'quality', q) ; % 利用 imwrite 函数完成压缩
        comdone = imread( 'temp. jpg') ;
        comdone = round( double( comdone) /255) ;
        for i = 1: M
              for j = 1: N
                    if comdone( i, j)  ~= image( i, j)
                          different = different + 1;
                    end
              end
        end
result( 1, count)  = different / ( M*  N) ;
different = 0;
end
plot( quality, result) ;
xlabel( 'jpeg 压缩率') ;
ylabel( '像素改变的百分比例') ;
title( '二值图像在 JPEG 条件下像素改变的状况')
```

分析的结果如图 5.23 所示。

分析得到，在压缩率大于 45% 时，图像基本上是鲁棒的。即使发生变化，像素的变化比例均不超过 1%！也就是说，λ取为 0.5% 就可以达到上述区分的目的。这里只讨论了 JPEG 压缩的影响，大家有兴趣可以结合后面水印测试和攻击的多种手段来分析一下像素改变的比例。

在 R_0，R_1 一定的情况下，λ设置过小是为了使每块图像块的像素都满足条件，要修改的像素就会多一些，同时，λ大一点，隐藏块与无用块的区别也大一些，提取出错的概率就小一些。综合考虑，在实验中，我们一般取 λ为 2%。大家在选用不同的载体实验时，建议先对 λ作简单的测试。

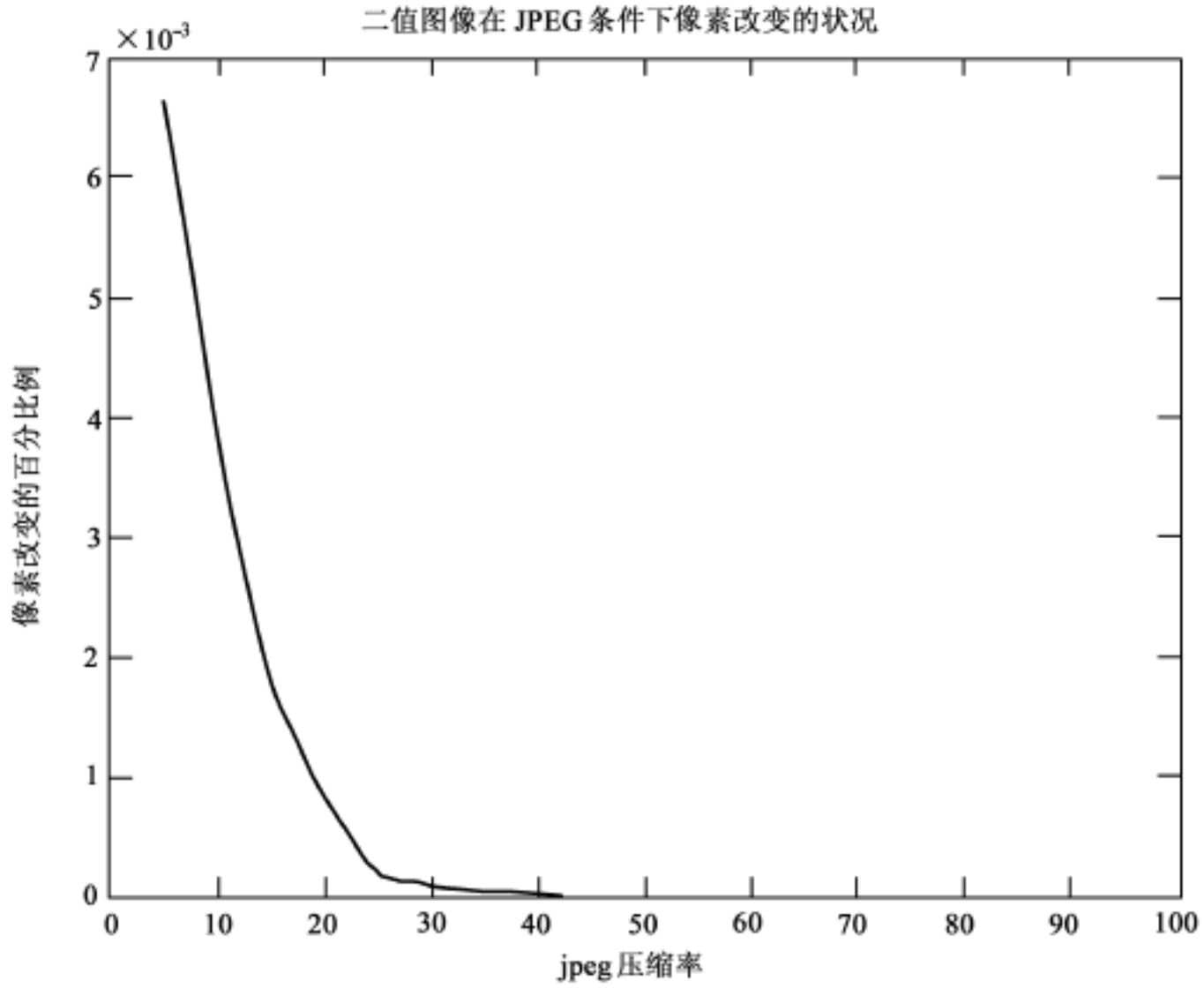


图 5.23　对 λ的讨论

(2)与隐藏容量的关系

表 5.1 列出了我们采用的 6 组实验数据。

表 5.1　　算法参数与隐藏容量的关系

R_0	R_1	λ	消息长度(bits)	分析过的块数	不可用块	难以调整块	不可用块占的比例
49	51	2	24	615	578	13	93.98%
48	52	2	24	588	550	14	93.53%
45	55	2	24	299	270	5	93.30%
*40	30	2	24	240	208	8	86.67%
45	55	3	24	293	258	11	88.05%
#45	55	5	24	209	173	12	82.78%

(* 所使用的参数有 2 块出现边界不足的情况,#所使用的参数有 7 块出现边界不足的情况)

二值的 lenna 图像是一个绝大部分黑白成片出现的图像，并不十分适宜本算法的发挥。当然，这并不影响我们对隐藏容量的分析。我们仅取 3 个字母 abc 为秘密信息(24bits)以保证载体足够大。以上五组数据均使用同样的密钥控制随机置乱和随机选择的结果，从数据中可以明显地看到，当取参数集{49, 51, 2}时，对全体图像 625 块，要分析 615 块才能确定 24bits 信息隐藏的位置，在这 615 块中，不可用块(图 5.16 情况 1)所占比例高达 93.98%。适当调整 R_0 和 R_1 的距离，可以略微改善这一状况。更重要的是，所分析的图像块的数量大大减小了。取参数集{45, 55, 2}时，隐藏 24bits 信息只用分析 299 块就确定了其隐藏的位置，从而也就可以认为将隐藏容量扩大了一倍!

健壮参数 λ与隐藏容量的关系似乎更紧密一些。当 R_0, R_1 一定时，λ的提高有利于隐藏容量的增大。当然，无限地扩大 R_0 与 R_1 的距离以及增大 λ将会导致边界不足的情况发生。在确保图像像素严格修改的情况下，取表 5.1 的第三组和第五组数据都是合适的。

(3)与隐藏不可见性以及鲁棒性的关系

图 5.24 给出了在表 5.1 下 5 组参数执行的情况。

图 5.24　二值隐藏的结果

从隐藏的直接效果上看，由于增大了 R_0 与 R_1 的距离以及 λ，需要修改的像素当然也更多，仔细观察，还是可以发现这一差异的。另外，图 5.23 已经表明了二值图像的鲁棒性是很强的，所以以上几组参数下的信息提取都是正确的。

5.3 基于图像其他特征的信息隐藏

在前面的小节中，我们已经对什么是空域信息隐藏进行了探讨。很显然，空域信息隐藏的一个突出特点就是将载体的冗余信号空间作为秘密信息的宿地址空间。通俗地说，在 RGB 颜色模型下，载体图像全部像素点的 LSB 就构成了一个冗余空间。图像载体是不是还有其他的冗余空间呢？答案是有的。

5.3.1 对图像亮度值的分析

在日常生活中我们处理一幅图像都毫无例外地涉及到了“图像亮度”这一概念（如图 5.25 所示）。与像素点表示的色彩一样，亮度的轻微改变同样是不易被人眼察觉的，在像素亮度值中找冗余就成为基于亮度的空域信息隐藏算法的首要内容。

图 5.25 日常生活中经常要对亮度进行处理

在第一章图像载体的基本知识中，我们曾简单地涉及到了亮度这一问题。一幅

图像的亮度是在相应的 YCbCr(YUV) 模型中的 Y 分量体现的。我们已经多次给出了 RGB 与 YCbCr 的转换关系:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.3316 & -0.50 \\ 0.50 & -0.4186 & -0.0813 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

可以看到, 一个像素 P(r, g, b) 的每一个分量都是在(0, 1) 区间的, 经过转换后得到的 Y 值同样也是在(0, 1) 之间的。在第一章的 HSV 模型中, 我们也曾谈到亮度这一问题。一般地说, 对于多数图像, 128 种颜色(H) 、8 种色饱和度(S) 和 16 种明度(V) 就足够了, 而饱和度和明度共同构成了通常意义下的亮度(参见 1.5.1) 。尽管图像灰度有 256 级, 但考虑到隐藏鲁棒性及实验条件的限制, 我们作出了如下的定义。

定义 5.1　我们取图像像素的原始亮度为 x 亮度。将其提高一个亮度即在其相应的 Y_x 值上加 $\frac{1}{8\times 16} = \frac{1}{128} = 0.0078$, 同理减一个亮度则是将其相应的 Y_x 值减去 $\frac{1}{128}$, 即将整个亮度空间划分成为 128 个单位。下面我们谈到的亮度都在这个数值范围内(如图 5.26 所示) 。

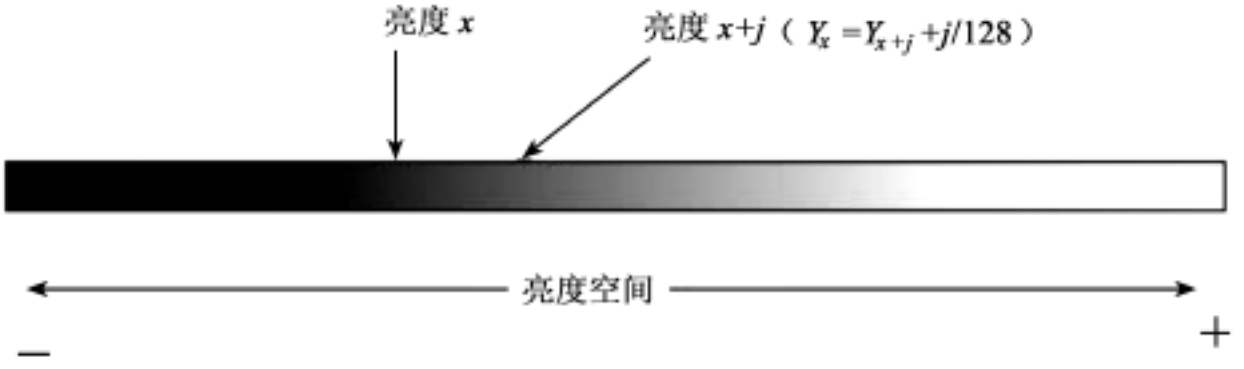


图 5.26　我们定义的亮度空间

接下来我们具体地改变一幅图像的亮度, 直观地体验一下亮度空间中的冗余。编写函数 brightanalysis. m 完成实验。函数 brightanalysis. m 的功能是根据要求改变输入图像相应像素点所对应的亮度值。这里需要调用第二章中的伪随机置换部分提供的函数 hashreplacement. m。该函数的具体内容请参见 2.6.3 节。

```
% 文件名: brightanalysis. m
% 程序员: 郭迟
% 编写时间: 2004. 2. 25
% 函数功能: 本函数是一个分析函数, 用以分析图像像素亮度的变化
% 输入格式举例: result = brightanalysis( 'c: \lenna. jpg', - 3, 0. 5) ;
%  result = brightanalysis( 'c: \lenna. jpg', - 3) ;
% 参数说明:
```

```
% image 为待分析的原始图像
% degree 为要求增减的亮度度数
% percent 为要求处理的像素占全部像素的百分比
% result 为处理结果
function result = brightanalysis( image, degree, percent)
% 将图像转换为 YUV 颜色空间, 提取亮度分量值
a = imread( image) ;
a = double( a) /255;
YUV = rgb2ycbcr( a) ;
bright = YUV(:,:, 1) ;
% 如果要求有比例则调用随机置换选择函数进行按比例随机选择像素点, 否则为全部像% 素点
if nargin == 3
    [ row, col] = size( bright) ;
    selectquan = row* col* percent;
    % 随机选取像素
    % hashreplacement 函数的三个种子固定为 row, col, selectquan, 减少输入参数的个数
    [ row, col] = hashreplacement( bright, selectquan, row, col, selectquan) ;
    for i = 1: selectquan
        bright( row( i) , col( i) ) = bright( row( i) , col( i) ) + degree /128; % 取 1 /128为亮度的 1 度
    end
% 未输入 percent 则对图像全体进行亮度处理
else
    percent = 1;
    bright = bright + degree /128;
end
% 转换为 RGB 模型显示处理效果
YUV(:,:, 1) = bright;
result = ycbcr2rgb( YUV) ;
subplot( 121) , imshow( a) , title( '原始图像') ;
subplot( 122) , imshow( result) ;
title( [ '随机调整', int2str( percent* 100) , '% 像素点亮度(', int2str( degree) , ')后的图像'] ) ;
```

利用 brightanalysis 函数, 我们获得了:

①将 lenna 全部像素点亮度 +1;

②对 lenna 随机选取一半的像素点将其亮度 - 1;

③对 lenna 随机选取一半的像素点将其亮度 +5;

④对 lenna 随机选取一半的像素点将其亮度 + 128 (显然超出了我们定义的范围) 。

四种条件下的实验结果如图 5. 27 所示。可以看到, 对亮度改变 1 个单位是不容易被察觉的。

图 5. 27　亮度的改变

5.3.2　基于图像亮度的信息隐秘示例

通过上一节的实验, 我们已经很清楚地认识到对图像亮度的轻微改变与对像素值的轻微改变一样, 都较难引起我们在视觉上的敏感。这就在空域隐藏中给我们提

供了非常现实的秘密信息隐藏空间。与 LSB 算法不同的是,我们并不是将秘密信息直接保留在载体的冗余空间上,而是通过一种特定的算法将亮度改变与秘密信息构成映射。

秘密信息的第 i bit 对应的隐藏位为 H_i

```
if ( 秘密信息的第 i bit  == 1)
        the lum of H_i += 1 or 2 degree;
else if ( 秘密信息的第 ibit  == 0)
        the lum of H_i -= 1 or 2 degree;
```

与这种隐藏算法对应的消息提取算法是要求原始图像(original) 参与的。用伪 C 代码描述如下:

```
从 image 中提取所隐秘的消息
if( the lum of H_i in image <  the lum of H_i in original )
        message_i = 0;
else
        message_i = 1;
```

秘密信息的第 i bit 对应的隐藏位 H_i 仍然通过第二章提供的伪随机选取策略来确定。由于在前面我们已经大量使用了伪随机置换方法,这里我们使用伪随机间隔法来完成实验,相应的将调用随机间隔控制函数 randinterval. m,该函数的相关内容请参见 2. 6. 1 节。编写函数 lumhide. m 与 lumextract. m 分别对应信息的隐秘和提取验证。

隐秘函数代码如下:

```
% 文件名: lumhide. m
% 程序员: 郭迟
% 编写时间: 2004. 2. 25
% 函数功能: 本函数将完成亮度空间下的信息隐秘
% 输入格式举例: [ result, count] = lumhide ( 'c: \lenna. jpg', 'c: \secret. txt',
'c: \test. png',1983, 2)
% [ result, count] = lumhide( 'c: \lenna. jpg', 'c: \secret. txt', 'c: \test. png',1983)
% 参数说明:
% cover 为载体图像
% msg 为秘密消息
% goalfile 为保存的结果
% key 为隐藏密钥
% scale 为实验中使用的调整亮度度数,默认为 1
% result 为隐藏结果
% count 为隐藏的信息数
```

```
function [ result, count] = lumhide( cover, msg, goalfile, key, scale)
% 默认的对亮度的调整为 1 度
if nargin ==4
    ascale = 1;
else
    ascale = scale;
end
% 按位读取秘密信息
frr = fopen( msg, 'r' ); % 定义文件指针
[ msg, count] = fread( frr, 'ubit1' ); % msg 为消息的位表示形式, count 为消息的 bit 数
fclose( frr) ;
% 读取载体图像信息, 并提取亮度分量
image = imread( cover) ;
image = double( image) /256;
YUV = rgb2ycbcr( image) ;
bright = YUV(:,:, 1) ;
% 调用伪随机间隔函数, 确定信息隐藏位
[ row, col] = randinterval( bright, count, key) ;
% 调整亮度进行隐藏
degree = ascale /128;
for i = 1: count
    if msg( i, 1) == 0
        bright( row( i) , col( i) ) = bright( row( i) , col( i) ) - degree;
    else
        bright( row( i) , col( i) ) = bright( row( i) , col( i) ) + degree;
    end
end
% 重构图像并写回保存, 建议使用 png 格式
YUV(:,:, 1) = bright;
result = ycbcr2rgb( YUV) ;
imwrite( result, goalfile, 'BitDepth','16) ; % 使用 png 格式, 以 16 位方式存储
subplot( 121) , imshow( image) , title( '原始图像') ;
subplot( 122) , imshow( result) ;
title( [ '取操作尺度为',int2str( ascale) , '下的信息',int2str( count) , 'bits 隐秘效果'] ) ;
```

信息提取函数代码如下:

```
% 文件名: lumextract. m
% 程序员: 郭迟
% 编写时间: 2004. 2. 25
% 函数功能: 本函数将完成亮度空间下的隐秘信息的提取
% 输入格式举例: result = lumextract( 'c: \lenna. jpg', 'c: \test. png', 'c: \extract. txt',
128, 1983)
% 参数说明:
% cover 为原始载体图像
% stegocover 为隐藏有信息的秘密消息
% goalfile 为信息提取后保存的地址
% key1 为秘密信息的 bit 数, 作为一个密钥参与计算
% key2 为提取密钥
% result 为提取信息
function result = lumextract( cover, stegocover, goalfile, key1, key2)
% 读取原始载体图像信息, 并提取亮度分量
originalimage = imread( cover) ;
originalimage = double( originalimage) /255;
originalYUV = rgb2ycbcr( originalimage) ;
originalbright = originalYUV(:,:,1) ;
% 读取隐蔽载体图像信息, 并提取亮度分量, 该载体应为 16 位存储方式的图像,
建议使用 png 格式
stegoimage = imread( stegocover) ;
stegoimage = double( stegoimage) /65535;
stegoYUV = rgb2ycbcr( stegoimage) ;
stegobright = stegoYUV(:,:, 1) ;
% 调用伪随机间隔函数, 确定信息隐藏位
[ row, col] = randinterval( stegobright, key1, key2) ;
% 准备提取并回写信息
frr = fopen( goalfile, 'a') ; % 定义文件指针
for i = 1: key1
    if originalbright( row( i) , col( i) ) > stegobright( row( i) , col( i) )
        fwrite( frr, 0, 'bit1') ; % 回写 0
        result( i, 1) = 0;
    else
        fwrite( frr, 1, 'bit1') ; % 回写 1
```

```
            result( i, 1)  = 1;
        end
    end
    fclose( frr) ;
```

图 5.28 是在不同的输入参数下得到的实验效果图。结合上一节中我们对亮度的分析可以知道，在 lumhide. m 函数中，输入参数 scale(操作尺度) 不宜太大(一般取 1 或 2) ，否则将影响信息隐藏的不可见性。

需要说明的是: 限于 MATLAB 自身的某些功能上的缺陷，我们在这里对使用的载体图像类型不作限制，但对隐蔽载体图像(stego_cover) 要求使用 png 格式。这是因为: 在实验中我们发现 MATLAB 的 imread 和 imwrite 函数(图像数据的读写函数) 对图像的默认操作都是 8 位的，一个图像的像素数据经过图像写回保存(imwrite) 和提取时的图像数据读取(imread) 两个操作后，由于多次的数据精度转换，原调整好的亮度值发生较大的偏移，其误差足以改变我们对一个像素亮度调整值的大小，从而导致实验失败(隐藏的信息无法正确读取) 。png 格式的图像允许以 16 位格式进行存储和读取，由于其像素占用的位数较大，精度较高，可以有效地避免 imread 和 imwrite 函数对数据的影响。

图 5.28　基于图像亮度的信息隐秘

5.3.3 基于图像亮度统计特性的数字水印

(1) Patchwork 数字水印算法

本节中,我们将简单地理解一下空域信息隐藏的一个著名算法: Patchwork 算法。Patchwork 算法是一种数据量较小、能见度很低、鲁棒性很强的数字水印算法,其生成的水印能够抗图像剪裁、模糊化和色彩抖动。“Patchwork”一词原指一种用各种颜色和形状的碎布片拼接而成的布料,它形象地说明了该算法的核心思想,即在图像域上通过大量的模式冗余来实现鲁棒数字水印,所以, Patchwork 算法也被认为是一个扩散被嵌入信息的典型代表。与 LSB 算法不同, Patchwork 是将水印信息隐藏在图像数据的亮度统计特性中,给出了一种原始的扩频调制机制。尽管该算法一般只能隐藏 1bit 信息,但仍然可以在一定程度上对图像数据的版权给予保护。必须强调的是:我们不能以隐写术的思想去理解数字水印,二者在使用领域上有着很大的不同。

以隐藏 1bit 数据为例, Patchwork 算法首先通过伪随机数生成器产生两个随机数据序列,分别按图像的尺寸进行缩放,成为随机点坐标序列,然后将其中一个坐标序列对应的像素亮度值降低,同时升高另一坐标序列对应的像素亮度。由于亮度变化的幅度很小,而且随机散布,并不集中,所以不会明显影响图像质量。我们所选取的伪随机数生成器的种子就是 Patchwork 算法的密钥。

Patchwork 的水印嵌入算法的具体描述如下:随机选择 N 对像素点(a_i, b_i),然后将每个 a_i 点的亮度值加 1 度,对每个 b_i 点的亮度值减 1 度,通过这一调整来隐藏信息,这样整个图像的平均亮度保持不变。用伪 C 代码描述为:

```
随机选择 N 对像素点(a_i, b_i)
lum( ) = 像素点亮度值
for ( i = 1; i < = N; i ++ )
        ( lum( a_i) , lum( b_i) ) = ( lum( a_i) + 1, lum( b_i) - 1) ;
```

上述算法基于一个基本的假设:给一个足够大的 n 值,对于根据伪随机数生成器生成序列选取的图像像素对(a_i, b_i),所有像素点 a_i 的亮度平均值与所有像素点 b_i 的亮度平均值非常接近。当对图像按 Patchwork 算法嵌入水印后,使得所有像素点 a_i 的亮度平均值增加 1,而所有像素点 b_i 的亮度平均值减少 1。在水印被嵌入后,这些像素点的亮度变化是能够被准确检测到的。这个假设是必要的且在水印嵌入和检测过程中可得到证实。

水印的检测算法与秘密信息的提取算法不同,不要求原始图像的参与,而仅根据待测图像来鉴别。其思想为:接受者计算 n 个 i 值的$(lum(a_i) - lum(b_i))$,如果这些亮度值之和 sum 接近于 2 的整数倍,则此水印可被检测出;但如果这些亮度值的和接近于 0,则此水印不能被检测出。通常我们根据经验选取一个适当的阈值 δ来决定数值的近似程度。注意,在本节中,这里的 2 表示的是 2 度亮度,对应数值应该为 1/64。算法的伪 C 代码描述为:

```
int k;
float δ
for ( i = 1; i <= N; i ++ )
        sum += ( lum( a_i ) - lum( b_i ) ) ;
if ( |sum - 2K| < δ  /* 此和值足够接近于 2k* /。
        存在水印;
else
        不存在水印;
```

下面我们讨论一下 Patchwork 算法的特性及其效果。前面已提到 Patchwork 水印嵌入量为 1bit, 现在我们来讨论 Patchwork 水印的不可见性。这种性质不容易概括, 主要因为这种性质取决于不同类型的图像的应用。上面算法中提到根据伪随机数生成器所选取的每一个像素的亮度值仅仅改变一个单位量(忽略一个像素的亮度值被多次改变) , 这意味着水印将难以被察觉。当算法应用于 8 bit 单精度图像时, 以上观点的正确性是毫无疑问的。当应用于二值图像时则存在问题。总的来说, 对大多数图像应用 Patchwork 水印算法是不易察觉的、安全的, 但 Patchwork 技术有其本身固有的局限性。

第一, Patchwork 技术的信息嵌入率非常低, 嵌入的信息量非常有限, 通常每幅图只能嵌入一个比特的水印信息, 这就限制了它只能应用于低水印码率的场合。因为嵌入码低, 所以该算法对串谋攻击抵抗力弱。为了嵌入更多的水印信息, 可以将图像分块, 然后对每一个图像块进行嵌入操作。

第二, 这种算法必须要找到图像中各像素的位置, 在有仿射变换存在的情况下, 就很难对加入水印的图像进行解码。尽管有这些应用的限制, 在不知道随机数字水印密钥的情况下, 要想移除数字水印仍是极其困难的, 除非破坏图像的视觉质量。

分析 Patchwork 水印算法的鲁棒性, 一个很明显的发现就是: 任何基于改变图像像素点位置的攻击都会使水印难以被检测出来。旋转、剪切尺度改变将会毁灭水印, 更甚的是任何基于改变像素点值的攻击很可能也会摧毁水印, 如滤波、锐化、有损压缩等, 所以, 嵌入 Patchwork 水印的图像将容易受到各种综合攻击的影响。为了增加水印的鲁棒性, 我们可以将像素对扩展为小块的像素区域(如 8× 8 图像块) , 增加一个区域中的所有像素点的亮度值而相应减少对应区域中所有像素点的亮度值。适当地调整参数后, Patchwork 方法对 JPEG 压缩、FIR 滤波以及图像裁剪有一定的抵抗力且人眼无法察觉。

影响 Patchwork 算法使用效果的因素很多, 主要有:

1) Patch 的深度

Patch 的深度是指对随机点邻域灰度值改变的幅度, 深度越大, 水印的鲁棒性越强, 但同时也会影响隐蔽性, 提高能见度。

2) Patch 的尺寸

大尺寸的 Patch 可以更好地抗旋转、位移等操作，但尺寸的增大必然会引起水印信息量的减少，造成 Patch 相互重叠。具体应用时必须在 Patch 的尺寸和数量两者之间进行折中。

3) Patch 的轮廓

具有陡峭边缘的 Patch 会增加图像的高频能量，虽然这有利于水印的隐藏，但也使水印容易被有损压缩所破坏。相反，具有平滑边缘的 Patch 可以很好地抗有损压缩，但易于引起视觉注意。合理的解决方案应该是在考虑到可能会遭受的攻击后确定，如果面临有损压缩的攻击，则应采用具有平滑边缘的 Patch，使水印能量集中于低频；反之，如果面临对比度调整的攻击，则应采用具有陡峭边缘的 Patch，使水印能量集中于高频。如果对所面临的攻击没有准确的估计，则应使水印的能量散布于整个频谱。

4) Patch 的排列

Patch 的排列应尽量不形成明显的边界，因为人眼对灰度边界十分敏感，W·Bender建议采用随机的六角形排列。

5) Patch 的数量

Patch 的数量越多，解码越可靠，但这同时也会牺牲图像的质量。

6) 伪随机序列的随机性能

算法采用的伪随机数生成器所生成的伪随机序列的随机性能当然地也决定数字水印的性能。

(2) 简单的 Patchwork 实验

针对前面的算法，我们曾设计了这样的一个实验：对 lenna 图像加 1bit 的 Patchwork 水印。对于两组随机序列的选择，我们先设计了这样一个算法：种子 key1 决定序列$\{a_i\}$，在奇数列像素点上进行随机间隔选取。种子 key2 决定序列$\{b_i\}$，在偶数列像素点上进行随机间隔选取。我们原本以为这样可以保证两组序列选出的像素点的平均亮度大体一致，但事实上这种做法严重地破坏了图像原有的统计特性，在局部(奇列像素或偶列像素)实现了随机控制，但在整体上大大加入了非自然因素，导致图像在加入水印后出现了明显的条状亮度分布，如图 5.29 所示。

图 5.29　明显的条状亮度分布

鉴于上述实验的教训，我们改用伪随机置换策略选取足够多的像素点，然后取其前一半构成像素集合$\{a_i\}$，另一半构成像素集合$\{b_i\}$。由于 hashreplacement. m 函数所选出的像素点具有不碰撞和较好的伪随机的特点，故而可以很好地解决选择两组像素这一问题。每组像素点的个数我们取为图像总像素点的 1/8。

亮度的调整策略与前面基于亮度的空域隐秘一样能完成水印信号的嵌入。同样地，也是由于 MATLAB 在数据精度处理上的缺陷，在水印检测中，我们不能严格按照上一小节的算法编写相应的程序。在前面给出的标准的 Patchwork 水印检测算法中，主要的判定式是：

$$| sum - 2k | < \delta$$

该式的实际意义是说两组经过调整的像素的平均亮度差值应该非常接近于 2 度。δ本身应该是一个比较小的数，但在理论上不应该小于原始图像未加水印前的平均亮度差值。只有当我们假定原始图像未加水印前的平均亮度差值为 0 时，才可以无限地将 δ取小以保证在水印检测中不出现将无水印的图像判断为有水印的图像这一错误。但在实际操作中，这样一来的直接结果就是导致了在水印检测中发生将有水印的图像判断为无水印的图像错误的概率大大增加，事实上这种错误是我们更应该去避免的，所以，我们在实验中调整判断方法为：

$$\frac{\sum_{i=1}^{quantity}(a_i - b_i)}{quantity} - 2 \text{lum} \approx \delta$$

quantity 为每组像素点的个数。对于加有水印的图像来说，$\{a_i\}$ 与 $\{b_i\}$ 的平均亮度差，从理论上说应该等于未加水印前的同样这些像素点的平均亮度差加上我们人为调整的 2 度亮度，所以，将判定阈值 δ设定为一个略大于原始图像未加水印前的平均亮度差的值，并考虑到 MATLAB 在数据处理中的误差，水印的判定式就成了：

$$\frac{\sum_{i=1}^{quantity}(a_i - b_i)}{quantity} - 1 \text{lum} > \delta$$

通俗地说，就是两组像素点的平均亮度差要比判定阈值 δ大 1 度多，就能表示其中含有水印信号。

表 5.2 是对于未加水印的原始 lenna 图像在不同种子控制下 $\{a_i\}$ 与 $\{b_i\}$ 的平均亮度差。$\{a_i\}$ 与 $\{b_i\}$ 的选取使用 hashreplacement. m。可以看到，在未加水印前，图像像素的平均亮度差远小于 1 度（1 度 = 1/128 = 0.0078）。在检测中，可以定义判定阈值 δ为 0.002 ~0.005。

表 5.2　　平均亮度差

种子 key1, key2, key3			$\{a_i\}$ 与 $\{b_i\}$ 的平均亮度差
1983	1121	421	0.0016581
2001	3253	11	0.00029515
27	8734	5608	0.00131
110	119	112	0.00074515
806	572	413	0.0021371

编写函数 patchworkwm. m 与 patchdetect. m 分别实现水印嵌入与检测的功能。

水印嵌入函数:

```
% 文件名: patchworkwm. m
% 程序员: 郭迟
% 编写时间: 2004. 2. 26
% 函数功能: 本函数将完成对图像加入 patchwork 水印
% 输入格式举例: result = patchworkwm( 'c: \lenna. jpg', 'c: \test. png',1983, 1121, 421, 2)
%  result = patchworkwm( 'c: \lenna. jpg', 'c: \test. png',1983, 1121, 421)
% 参数说明:
% original 为原始图像
% goalfile 为保存的结果
% key1, key2, key3 为序列密钥
% scale 为实验中使用的调整亮度度数, 默认为 1
% result 为加入水印后的结果
function result = patchworkwm( original, goalfile, key1, key2, key3, scale)
% 默认的对亮度的调整为 1 度
if nargin == 5
    ascale = 1;
else
    ascale = scale;
end
% 读取图像信息, 并提取亮度分量
image = imread( original) ;
image = double( image) /256;
YUV = rgb2ycbcr( image) ;
bright = YUV(:,:, 1) ;
% 定义两组像素点的个数
[ m, n] = size( bright) ;
quantity = floor( m* n /8) ;
% 调用伪随机置换函数, 确定信息隐藏位
[ row, col] = hashreplacement( bright, 2* quantity, key1, key2, key3) ;
% 调整亮度
degree = ascale /128;
for i = 1: quantity
    bright( row( i) , col( i) ) = bright( row( i) , col( i) ) + degree;
```

```
        bright( row( 2* i) , col( 2* i) ) = bright( row( 2* i) , col( 2* i) ) - degree;
    end
    % 重构图像并写回保存, 建议使用 png 格式
    YUV(:,:, 1) = bright;
    result = ycbcr2rgb( YUV) ;
    imwrite( result, goalfile, 'BitDepth',16) ;
    subplot( 121) , imshow( image) , title( '原始图像') ;
    subplot( 122) , imshow( result) ;
    title( [ '取操作尺度为',int2str( ascale) , '下嵌入 patchwork 水印的效果'] ) ;
```

与图 5.28 一样, 图 5.30 是取不同的操作尺度下的水印嵌入图像, 但操作尺度取得过大是会影响水印不可感知性的。在图 5.30 中, 我们有意选择了一个严重超出范围的操作尺度 128 对图像添加水印, 从该图像结果上可以很明显地看到许多“白点”和“黑点”, 它们就分别对应了算法中的$\{a_i\}$和$\{b_i\}$。

图 5.30　Patchwork 数字水印

水印检测函数:

```
% 文件名: patchdetect. m
% 程序员: 郭迟
% 编写时间: 2004. 2. 26
% 函数功能: 本函数将完成对图像加入 patchwork 水印
% 输入格式举例: result = patchdetect( 'c: \test. png',1983, 1121, 421, 0. 001) ;
% 参数说明:
% test 为待测的图像
% key1, key2, key3 为序列密钥
% threshold 为判断阈值
% result 为检测的结果
function [ result, cmpvalue] = patchdetect( test, key1, key2, key3, threshold) ;
```

```
% 读取隐蔽载体图像信息,并提取亮度分量,该载体应为 16 位存储方式的图像,建议使用 png
image =  imread( test) ;
image = double( image) /65535;
YUV = rgb2ycbcr( image) ;
bright = YUV(:,:, 1) ;
% 求两组像素点的个数
[ m, n]  = size( bright) ;
quantity = floor( m*  n /8) ;
% 调用伪随机间隔函数,确定信息隐藏位
[ row, col]  = hashreplacement( bright, 2*  quantity, key1 , key2, key3) ;
% 求 sum 值
sum = 0;
for i = 1: quantity
        sum = sum + bright( row( i) , col( i) )  -  bright( row( 2*  i) , col( 2*  i) ) ;
end
% 与阈值进行比较
cmpvalue = abs( sum/quantity)  - 1/128;
if cmpvalue > threshold
        result = 1;
        disp( 图像含有水印信号 ) ;
else
        result = 0;
        disp( 图像不含有水印信号 ) ;
end
```

以上便完成了对 Patchwork 水印的一个简单模拟。实际应用中的 Patchwork 算法是比这里的例子复杂的。对于 Patchwork 水印的攻击和性能检测等问题,我们将在后面数字水印的相关章节集中进行探讨。

考虑到灰度图像实际上就是 YUV 模型中的 Y 分亮,所以,基于亮度统计特性的空域隐藏算法比较多地应用在灰度图像上,当然这并不表示它们只适用于灰度图像。

5.4 文本载体的空域信息隐藏

目前,对数字水印技术的研究主要集中在静止图像和音频、视频方面。事实上,有很多的文本信息和图像、视频信息一样需要得到保护(例如遗嘱等) 。我们在这里仅向大家介绍一些学者所提出的思想和方法,供大家参考。

5.4.1　嵌入方法

最原始的文件(如 ASCII 文本文件)由于不存在可插入标记的可辨认空间,因而不能直接用来插入水印。对于一些高级形式的文本文档,由于它们通常都是格式化文档(如 PDF, DOC 等),因此可以在版面布局或格式化的编排上做文章。将某种变化视为数字 1,不变化视为 0,这样嵌入的数字水印信号就可以被认为是具有某种分布形式的伪随机序列。

以一个英文的文本文件为例,由于其是由字母按照某种行、列、段落结构所组成的,我们可以考虑对它做细微的改动,而不引起视觉上的差异。

(1)行移编码

对于大部分的文档文件,在同一个段落内的各行的间距是均匀的,将文本的一整行做垂直的移动即可作为嵌入信息的标记,当某一行做上移或下移时,其相邻的一行或两行保持不动,作为在解码时的参考位置。当垂直位移量小于或等于 1/300 英寸时肉眼将无法辨认。在解码的过程中,我们无需根据文本的首末附加信息来判断嵌入信息与否,而只需判断其行间距。

(2)字移编码

在格式化的文档中通常使用变化的单词间距,我们可以水平移动某一个单词来标记插入的信息,而使其左右的单词保持不动。因为人眼无法辨认 1/150 英尺以下的单词水平位移量,所以,这种方法同样能达到效果。由于初始的单词间距是不均匀的,在解码时是需要原始文档参加的,因此,这种方法只适用于已知原始文档的隐秘。

(3)特征编码

在文档中的每个字母都会有其特征,如字母的高度、宽度等,特征编码即通过改变这些特征来加入标志信息。我们可以改变某一字母的高度,在解码时通过比较同一页中没有改变高度的字母来恢复隐秘信息。但是,如果有一个特征没有改变的字母与其相邻,读者是很容易看出它们之间的区别的,因此,用这种方法必须十分的细心。在检测时是否需要原始文档的参加是根据改变的方法来确定的。

5.4.2　文本水印的检测

在现实生活中,因为文本文档最终仍是以纸质形式传播的,所以对文本水印的检测实际上是对文本图像中的水印进行检测。所谓“文本图像”,就是指存放文字内容的图像。如某一份遗嘱,经过在文本编辑器上调整格式编码而嵌入水印后,再经过打印、若干次的复印等形成纸质文档。最终要验证是否有水印的正是这些在人们手中传播的纸质文档,所以验证之前往往通过扫描仪将遗嘱文本扫描到计算机中,以一幅图像的形式存放,这幅图像就是文本图像。

首先,在对文本图像进行检测之前必须对其进行相应的预处理,因为文本图像一般都是由一些设备(如打印机、复印机、扫描仪等)再生的,而这些设备都可以看成是

一个有噪信道,产生的噪声可以看成是椒盐噪声,因而采用中值滤波的方法,且根据字体的特征等定义不同大小的模板。

其次,需要调整斜度。所谓斜度是指文本图像在扫描或复印过程中产生的图像倾斜。在检测斜度时普遍采用的方法是利用投影分布图的重复计算获得倾斜角度。一些文字识别软件(如汉王尚书文字识别软件等)提供有斜度校正的功能。

(1)对行移和字移的检测

对行移和字移的检测方法是 Steven 和 Maxemchuk 提出的,主要是通过创建并分析一个页面图形的映射轮廓来检测水印的。一个页面图形数字化以后是一个二维的数组:

$$f(x, y) \qquad x = 0, 1, \cdots, W \quad y = 0, 1, \cdots, L$$

其中,$f(x, y)$ 表示在坐标(x, y)处的像素强度,我们知道,对于一个黑白图像,$f(x, y)$ 的取值是从 0 到 1 的,而 $W \times L$ 是像素图形的大小,由扫描结果决定。每一个数组行对应扫描结果的一个水平行,由此我们可以得到一个包括一个单独文本行的子图像为:

$$f(x, y) \qquad x = 0, 1, \cdots, W \quad y = t, t + 1, \cdots, b$$

其中,t 和 b 分别表示这一行的最上方和最下方的像素行坐标。轮廓定义为一个二维数组到一维的映射,一个文本行的子数组的水平轮廓为:

$$h(y) = \sum_{x=0}^{w} f(x, y) \qquad y = t, t + 1, \cdots, b$$

这种水平轮廓有明显的“柱”与“谷”,“柱”对应于文本行,“谷”则对应于两行间的空白,而柱的宽度对应于这一行字母本身的高度,由此可以检测到行移和字移。

(2)特征检测

通过辨认轮廓中柱的特征的所在位置可以识别该柱。对于英文文本,每个柱有两个明显的峰值,这些峰值对应的是扫描线通过的文本行中间线和基线。左边的峰产生于字母的中间线,即诸如字母 A,e 中间都有的那一个水平线。右边的峰对应于字母的“脚”,也就是基线,通过基线可以获得行间距,这样对于那些最初行间距固定的文本,就可以不需要最初的文档轮廓而检测到行移了。

(3)相关检测

相关检测器在加性高斯白噪声存在的情况下可以最佳检测信号。文本通过复印、扫描等过程叠加的噪声可以认为主要是加性高斯白噪声,把文章轮廓看做是离散时间信号,将接收图形轮廓 $g(y)$ 作为接收信号,而初文本轮廓 $h(y)$ 作为发射信号,如果设服从 $N(0, \sigma^2)$ 的加性高斯白噪声为 $N(y)$,有 $g(y) = h(y) + N(y)$,通过相关检测器,用最大似然检测法则直接判断出行移和字移最有可能的移动方向。

(4)质心检测

水平轮廓中含有明显的高而窄的柱,可以采用一个柱的质心处的坐标近似地表示该柱。从单个文本行的垂直轮廓中可以看出,单词之间的间距远大于同一个单词

的字母之间的间距，因此，可以用单词的质心坐标近似地表示单词的轮廓，字移和行移的规则意味着在中间块的质心轻微移动的同时，还要保持两个控制块的质心不变。质心检测的判断依据是中间块的质心相对相邻的两个控制块的质心距离的变化。

假设有分别定义在$[b_1, e_1]$，$[b_2, e_2]$，$[b_3, e_3]$上的相邻的三个文本行，其中b_i是图像中第 i 个文本行纵坐标的起点，而e_i是纵坐标的终点，c_i是第 i 行质心的纵坐标。对中间文本行进行标记，将中间行移动量记为 ε，$\varepsilon>0$表示上移，$\varepsilon<0$表示下移。对于初始文本轮廓 h(y)，第 i 行的质心为：

$$c_i = \frac{\sum_{b_i}^{e_i} yh(y)}{\sum_{b_i}^{e_i} h(y)} \qquad i = 1, 2, 3$$

将轮廓 g(y) 中的质心位移用加性轮廓噪声 N(y) 来表示，这时参照行的质心为$\mu_1 = c_1 + V_1$，且$\mu_3 = c_3 + V_3$，因为中间行移动了 ε，所以它的质心为$\mu_2 = c_2 + V_2 - \varepsilon$，其中$V_i$表示原质心受加性轮廓噪声而产生的偏移，将受破坏的质心距离与未标记的质心距离的区别作为检测过程的判断变量。

$$\Gamma_u = (\mu_2 - \mu_1) - (c_2 - c_1)$$

$$\Gamma_d = (\mu_3 - \mu_2) - (c_3 - c_2)$$

Γ_u为中间行与上参照行间的质心距离变化，Γ_d为中间行与下参照行间的质心距离变化。当观察到的(Γ_u, Γ_d)的值为(γ_u, γ_d)时，最大似然判断准则是：

上移：$\gamma_u / v_1^2 \leq \gamma_d / v_3^2$

下移：反之

式中v_1^2, v_3^2是上参照行与下参照行的质心噪声方差，由下式给出：

$$v_i^2 = \frac{\delta_i^2 \omega_i}{H_i^2}\left[\delta_i^2 + (\omega_i^2 - 1)/12\right]$$

式中，$H_i = \sum_{b_i}^{e_i} h(y)$

$$\omega_i = e_i - b_i + 1$$

$$\delta_i = C_i - (e_i + b_i)/2$$

关于文本水印的研究并不深入，我们也只能作简单的引述，具体的应用有待大家来实现。