LA-UR-96- 1770

Title:

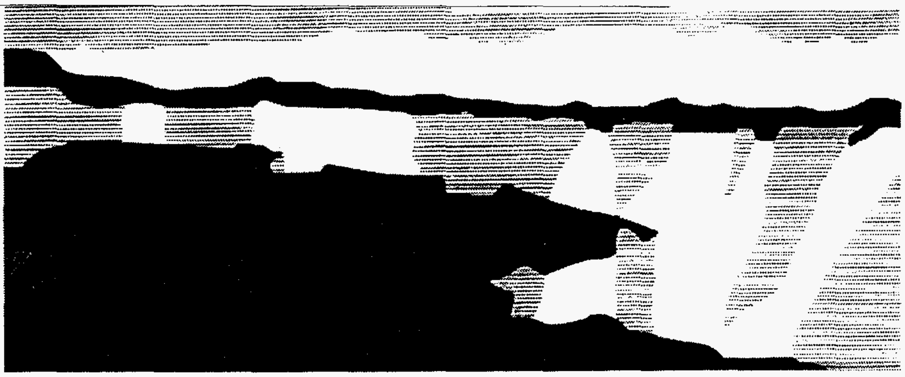
The Data Embedding Method

Author(s):
Maxwell T. Sandford, NIS-9
Jonathan N. Bradley, CIC-3
Theordore G. Handel, NIS-9

Submitted to: DOE Office of Scientific and Technical Information (OSTI)

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.



Los Alamos
NATIONAL LABORATORY

The data embedding method

Maxwell T. Sandford II, Jonathan N. Bradley, and Theodore G. Handel
Los Alamos National Laboratory, NIS-12
P.O. Box 1663, Los Alamos, NM 87545

ABSTRACT

Data embedding is a new steganographic method for combining digital information sets. This paper describes the data embedding method and gives examples of its application using software written in the C-programming language. Sandford and Handel[1] produced a computer program (BMPEMBED, Ver. 1.51 written for IBM PC/AT or compatible, MS/DOS Ver. 3.3 or later) that implements data embedding in an application for digital imagery. Information is embedded into, and extracted from, Truecolor or color-pallet images in Microsoft® bitmap (.BMP) format.

Hiding data in the noise component of a host, by means of an algorithm that modifies or replaces the noise bits, is termed 'steganography.' Data embedding differs markedly from conventional steganography, because it uses the noise component of the host to insert information with few or no modifications to the host data values or their statistical properties. Consequently, the entropy of the host data is affected little by using data embedding to add information. The data embedding method applies to host data compressed with transform, or 'lossy' compression algorithms, as for example ones based on discrete cosine transform and wavelet functions.

Analysis of the host noise generates a key required for embedding and extracting the auxiliary data from the combined data. The key is stored easily in the combined data. Images without the key cannot be processed to extract the embedded information. To provide security for the embedded data, one can remove the key from the combined data and manage it separately. The image key can be encrypted and stored in the combined data or transmitted separately as a ciphertext much smaller in size than the embedded data. The key size is typically ten to one-hundred bytes, and it is derived from the original host data by an analysis algorithm.

Keywords: data embedding, steganography, bandwidth expansion, auxiliary data, communication channel, covert communication, cipher

1. INTRODUCTION

Data embedding is a steganographic process for combining digital information sets. Data containing stochastic, or random, noise are used as a carrier or host for additional, unrelated information. Embedding manipulates the stochastic noise component of the host data. The stochastic noise is replaced with pseudo-random noise during the embedding process. Digital information separate from the host data are not correlated with the stochastic noise component, and are therefore a suitable source of pseudo-random variations. Thus, the noise component intrinsic to the host data becomes a channel for information transfer.

Data embedding utilizes two data properties common to nearly all data representing measurements made from nature. Redundancy and noise are properties present in many digital data sets. Data are redundant because the same values occur more than once in the stream of numerical values representing the information. Noise is found in any data produced by sensors. The numerical values are uncertain to within some value. The uncertainty is measured by the ratio of signal S to noise N. Thus, on average, the measured quantity represented by the numerical data value is S+N. The data embedding method consists of identifying and manipulating numerical values that are redundant to within the noise limit of the data set. Host data sets permit embedding about $(S/N)^{-1}$ additional information fraction. Thus for $S/N \approx 10$, about 10% of the carrier data can be replaced with embedded information. Data having no intrinsic noise component, computer instructions for example, are not usable as a carrier of embedded information. Most data result from signal sampling, and are therefore usable as hosts for embedded information.

2. APPLICATION OF METHOD

2.1 Analysis of greyscale-format image host data

The embedding process is conducted in two parts. First, an estimate of the noise component of the host data is used in combination with an analysis of the histogram of numerical values to identify pairs of data elements redundant or identical in value, and occurring with about the same statistical frequency. Second, the position of occurrence of the values found is adjusted to embed the bit-stream of another information set.

The principle of data embedding is re-arrangement of the host data values in order to encode the values of the data in a separate information set. The host data values are not changed by data embedding. Data embedding does not change low-order bits or create data values that are not found in the original host data.

As example consider host data represented by 8 bits of binary information. The values range between 0 and 256 for each host data sample. Assume the noise value for signal S is N = ± S/10, or approximately 10% of the signal value. For many data, the noise value can be approximated by assigning a constant value, and for the example following we take N = ± 10. Two values in the host data d_i and d_j are within the noise value if

$$|d_i - d_j| \quad = \quad \varepsilon \;\leq\; N. \qquad (1)$$

The frequency of occurrence of the value d_i is $f(d_i)$. Data values meeting the criteria in eq. (1), and occurring in the host data with frequency $|f(d_i)-f(d_j)| < \delta$, where δ is the tolerance imposed for statistical equality, are candidates for embedding. The values d_i and d_j constitute a pair of data values, p_k. The presence of one member of the pair is taken to represent an embedded 0-bit, and the other, a 1-bit. There are $k = 0,1,2,...N_p$ such pairs in the host data set giving a total number of embedding bits for each pair

$$M_k \;=\; \sum_i f(d_i) + \sum_j f(d_j) \;, \qquad (2)$$

where the summations for i and j run to the limits of the number of pairs found in the host data set.

2.2 Analysis of color pallet-format host image data

To illustrate the determination of the pair values in eq. (1), we use an example from our demonstration program for embedding data into digital image. The values d_i and d_j are the pixel values in the image. These 8-bit values are interpreted as indices in the color-pallet table. The comparison indicated in eq. (1) is therefore one between colors in the pallet. Entries in the color pallet are Red, Green, and Blue (RGB) color-component values. Additional information on the format used for Bitmap images can be found in the book by Levine[2] and in the article by Luse[3].

Identifying the pair values is illustrated by the code in Listing 1 (extracted from the BMPEMBED.C program). The code fragment begins with a loop running over the number of colors in the pallet (256, for an image having 8-bits per pixel). The loop index i is used to test each pallet colors against all the other entries, in sequence, to identify pairs of color entries meeting the criteria in eq. (1). Each color in the i-loop is tested against all other colors in the pallet by a second loop using index j, starting at line 16. Line 5 provides a modification for images having a pallet for greyscale instead of colors. For these images, the RGB components are identical for each pallet entry, although some formats include a 16-color table as well[3].

The comparison indicated in equation (1) is made by converting the Red, Green, and Blue (RGB) color component values to corresponding Hue, Saturation, and Intensity (HSI) color components. Line 12 uses a separate routine rgbhsi() to effect this conversion. Line 20 converts RGB color component values in the j-loop to HSI data-

2. APPLICATION OF METHOD

2.1 Analysis of greyscale-format image host data

The embedding process is conducted in two parts. First, an estimate of the noise component of the host data is used in combination with an analysis of the histogram of numerical values to identify pairs of data elements redundant or identical in value, and occurring with about the same statistical frequency. Second, the position of occurrence of the values found is adjusted to embed the bit-stream of another information set.

The principle of data embedding is re-arrangement of the host data values in order to encode the values of the data in a separate information set. The host data values are not changed by data embedding. Data embedding does not change low-order bits or create data values that are not found in the original host data.

As example consider host data represented by 8 bits of binary information. The values range between 0 and 256 for each host data sample. Assume the noise value for signal S is $N = \pm S/10$, or approximately 10% of the signal value. For many data, the noise value can be approximated by assigning a constant value, and for the example following we take $N = \pm 10$. Two values in the host data d_i and d_j are within the noise value if

$$|d_i - d_j| \quad = \quad \varepsilon \ \leq \ N. \tag{1}$$

The frequency of occurrence of the value d_i is $f(d_i)$. Data values meeting the criteria in eq. (1), and occurring in the host data with frequency $|f(d_i)-f(d_j)| < \delta$, where δ is the tolerance imposed for statistical equality, are candidates for embedding. The values d_i and d_j constitute a pair of data values, p_k. The presence of one member of the pair is taken to represent an embedded 0-bit, and the other, a 1-bit. There are $k = 0,1,2,\ldots N_p$ such pairs in the host data set giving a total number of embedding bits for each pair

$$M_k \ = \ \sum_i f(d_i) + \ \sum_j f(d_j) \ , \tag{2}$$

where the summations for i and j run to the limits of the number of pairs found in the host data set.

2.2 Analysis of color pallet-format host image data

To illustrate the determination of the pair values in eq. (1), we use an example from our demonstration program for embedding data into digital image. The values d_i and d_j are the pixel values in the image. These 8-bit values are interpreted as indices in the color-pallet table. The comparison indicated in eq. (1) is therefore one between colors in the pallet. Entries in the color pallet are Red, Green, and Blue (RGB) color-component values. Additional information on the format used for Bitmap images can be found in the book by Levine[2] and in the article by Luse[3].

Identifying the pair values is illustrated by the code in Listing 1 (extracted from the BMPEMBED.C program). The code fragment begins with a loop running over the number of colors in the pallet (256, for an image having 8-bits per pixel). The loop index i is used to test each pallet colors against all the other entries, in sequence, to identify pairs of color entries meeting the criteria in eq. (1). Each color in the i-loop is tested against all other colors in the pallet by a second loop using index j, starting at line 16. Line 5 provides a modification for images having a pallet for greyscale instead of colors. For these images, the RGB components are identical for each pallet entry, although some formats include a 16-color table as well[3].

The comparison indicated in equation (1) is made by converting the Red, Green, and Blue (RGB) color component values to corresponding Hue, Saturation, and Intensity (HSI) color components. Line 12 uses a separate routine rgbhsi() to effect this conversion. Line 20 converts RGB color component values in the j-loop to HSI data-structure components, and the following line calculates the color difference in the HSI system. Line 24 implements the test required by eq. (1) above. If the color difference is less than a fixed noise value (COLOR_NOISE = 10 in this example), the intensity difference is tested to see if the two pallet entries differ by less than the noise value (INTEN_NOISE = 10 in this example). Two additional constraints are imposed before accepting the entries as candidate pair values. The difference in color is required to be the smallest color difference between the test (i-loop) value and all the other (j-loop) values. The number of pairs selected (k) must be less than half the number of columns in a row of pixels in the image. The last constraint is algorithmic and is not required by the embedding process. It is imposed in order for the BMPEMBED program to store the pair values in a single row in the picture.

A data-structure array **pair[]** is used to hold the values of candidate pairs (i,j) and their total frequency of occurrence, M_k. If the image is a greyscale pallet, the test at line 35 is used to force comparing only the intensity of the two pallet entries. Greyscale images do not require the RGB-HSI conversion made for color pallets. The BMPEMBED algorithm ignores differences in the saturation component of color pallet entries, because saturation is ordinarily not noticeable in a color image. The Hue and Intensity components are constrained within fixed noise limits to determine the pallet pair values. In this example, for pallet-color images, the constraint for statistical equality of the pair members is ignored.

Listing 1

```
line 1      for(i = 0; i < (int)bh.colors; i++) {
              int avg;
              if(i % 10 == 0)fprintf(stderr,".");
              c1.red = colormap[i].r;
              c1.grn = colormap[i].g;
              c1.blu = colormap[i].b;
line 7                if(greyscale)  {
                              avg = (int)(c1.red + c1.grn + c1.blu)/3;
                              if(avg==0)continue;
                              if(avg!=c1.red || avg!=c1.grn || avg!=c1.blu)continue;
                              }
line 12       (void)rgbhsi(&c1, &d1);        /* convert to HSI components */
              if((int)d1.inten == 255)unused++;
              old_diff = 0.f;
              if((int)d1.inten==0 || (int)d1.inten==(int)bh.colors)continue;
line 16       for(j=i+1; j < (int)bh.colors; j++)          {
                      c2.red = colormap[j].r;
                      c2.grn = colormap[j].g;
                      c2.blu = colormap[j].b;
line 20               (void)rgbhsi(&c2, &d2);           /* convert to HSI components */
                      color_diff = d2.hue - d1.hue;
                      /* hue & inten. difference must be ok */
                      if(!greyscale)          {
line 24                 if((abs((int)color_diff) < COLOR_NOISE)\
                          && (color_diff < old_diff)\
                          && ((int)fabs((double)(d2.inten-d1.inten)) < INTEN_NOISE) )  {
```

Listing 1, continued

```
                                        if(k>(int)bh.cols/2 -1)break;
                                        pair[k].i = i;
                                        pair[k].j = j;
                                        pair[k].count = 0;
                                        k++;
                                        old_diff = color_diff;
                                        }
                              }
line 35                        else {
                                 avg = (int)(c2.red + c2.grn + c2.blu)/3;
                                 if(avg==0)continue;
                                 if(avg!=c2.red || avg!=c2.grn || avg!=c2.blu)continue;
                                 if( (int)fabs((double)(d2.inten-d1.inten)) < INTEN_NOISE &&
                                         (int)fabs((double)(d2.inten-d1.inten)) !=0)        {
                                               if(k>(int)bh.cols/2 -1)break;
                                               pair[k].i = i;
                                               pair[k].j = j;
                                               pair[k].count = 0;
                                               k++;
                                               }
                                 }
                              } /* j loop */
                        if(k>(int)bh.cols/2 -1)break;
                        } /* i loop */
line 51               no_pairs = k;
```

Pair values found by the code described above include generally redundant values. The same index value i can be found in several different pair combinations. Because multiple pairs can contain the same pallet entry, it is necessary to eliminate some pairs ensuring that unique combinations of pallet indices are used for embedding data bits. The number of pairs located by applying the criterion in eq. (1) is stored in the variable no_pairs, in line 51 above.

The code fragment in Listing 2 illustrates how duplicate pairs are eliminated by a separate routine. First the histogram of the image is used to calculate the total number of occurrences in each pair (eq. 2). Line 1 below starts the loop used to calculate the value M_k for each pair. Next, the pairs are sorted according to decreasing order of the **pair[].count** data-structure member (line 5, listing 2). The elimination of duplicates in the following line retains the pairs p_k having the largest total number of frequency values M_k. Line 10 and the following lines calculate the total number of bytes that can be embedded into the host data using the unique pairs found by the algorithm.

Listing 2

```
line 1          for(i=0;i<k;i++) {
                        pair[i].count += (hist_values[pair[i].i] + hist_values[pair[i].j]);
                        if(pair[i].i==0 || pair[i].j==0)pair[i].count = 0;
                        }
line 5          p_sort(pair, k);
                no_pairs = duplicate (k, pair);
                total = 0;
                for(i=0;i<no_pairs;i++)
                  total += pair[i].count;
line 10         total /= 8;
                value = (float)total - (float)NCOLS;
                if(value > 0.f) fprintf(stderr,"\n%.1f Kb embedding space located", value/1000.f);
                if(value == 0.f)fprintf(stderr,"\nNo embedding space available in this image");
                if(value < 0.f) fprintf(stderr,"\nInsufficient embedding space");
```

2.3 Analysis of Truecolor-format host image data

The algorithm described for pallet-format images manipulates pixel values without regard to the individual frequency of occurrence. In listing 3, for Truecolor images^2, we enforce the constraint on the frequency of occurrence that minimizes the effect of embedding on the host data histogram. Truecolor images consist of three individual 8-bit greyscale images, one each for the R, G, and B-image components. The combination of the three 8-bit components gives approximately 16 million colors. The BMPEMBED algorithm embeds data into Truecolor images by treating each RGB color-component image separately. The pixel values are used directly as components in the embedding pairs, because no pallet exists. The effect of embedding on the image color is therefore within the noise value of the individual intensity components.

In listing 3, the **ip**-loop starting in line 2 refers to the color plane (ip = 0,1,2 for R,G,B). The frequency of occurrence of each numerical value (0 through 255) is given in the array **hist_values[]**, with the color plane histograms offset by the quantity **ip*256** (see line 7 in listing 3). The variable **fvalue[]** holds the floating point histogram for color-component **ip**. Line 11 begins a loop to constrain the pairs selected to equal frequency of occurrence. Pixel intensities within the noise limit RANGE are selected for comparison of statistical frequency. The tolerance δ for statistical agreement is fixed at 5% in line 17. After all the possible values are tested for the constraints of noise and statistical frequency, the pairs found are sorted (line 27), duplicates removed, the starting index is incremented (line 31), and the search continues. A maximum number of pairs is set by the constraint that the i and j pair values be less than the number of pixels in an image row, taken as 256 for the example.

Listing 3

```
line 1     /* Find histogram point-pairs within RANGE counts, within 10% in number */
            for(ip=0;ip<3;ip++)            {
                   int nstart;
                   long ii;
                   fprintf(stderr,"\nAnalyzing intensity histogram for plane %d", ip);
line 6             for (i=0;i<256;i++) {
                     fvalue[i]=(float)hist_values[ip*256+i];                 }
                     }
```

Listing 3, continued

```
                    nstart = RANGE;
                    k = 0;
                    while(nstart<256 && k<(int)bh.cols/2)      {
line 12                  for (i=nstart;i<256;i++)          {
                              for(j=i-1;j>i-RANGE;j--) {
                                   li = hist_values[ip*256+i];
                                   if((int)(fvalue[j]*fvalue[i])==0)break;
                                   if((float)fabs((double)(fvalue[j]-fvalue[i]))\
line 17                                 < 0.05f*(fvalue[j]+fvalue[i]))     {
                                        pair[k].i = i;
                                        pair[k].j = j;
                                        pair[k].count = li + hist_values[ip*256+j];
                                        k++;
                                        if(k>(int)bh.cols/2-1)break;
                                        }
                                   } /* end of inner pixel comparison loop  (j) */
                              if(k>(int)bh.cols/2-1)break;
                              } /* end of outer pixel comparison loop (i) */
line 27                  p_sort(pair, k);
                         no_pairs = duplicate (k, pair);
                         k = no_pairs;
                         if(verbose)fprintf(stderr,"%3d pairs\r", k);
line 31                  nstart = i;
                         } /* end of while loop */
```

Applying the statistical constraint minimizes the effects of embedding data into the host data. If the tolerance δ = 0, each pair chosen will contain data values less than the noise value in separation and occurring with *exactly the same statistical frequency*. If random bits are embedded, the pair values will on average occur with the same statistical frequency after the embedding process, but in different sequence in the image. However, few if any pairs might be found having exactly the same frequency of occurrence. The statistical frequency tolerance of 5% used in the algorithm shown in listing 3 gives more pairs that are close in frequency, and preserves most of the statistics of the original host data.

3. THE DATA EMBEDDING ALGORITHM

3.1 Embedding into pallet-color images

Embedding data into host information consists of re-arranging the order of occurrence of redundant numerical values. The positions of the host data value pair-components found by analysis encode the bit-stream into the host. The values used are the ones occurring in the host. Embedding *does not change the numerical values in the host data.*

The nature of the embedded information has little effect on the embedding result, except for the obvious recognition that embedding a repetitive bit pattern modulates the host data noise component so it might become noticeable[†] . Encrypting or compressing the auxilary data prior to embedding randomizes the bit stream sufficiently that embedded data are not distiguishable easily from the original stochastic noise. A variant of embedding (Section V) can be used to impress a removable digital watermark on the host data.

In the BMPEMBED algorithm, the host data are processed sequentially. The first pass through the data examines each value and tests for a match with the values in an embedding pair. Matching values in the host data are initialized to the data-structure value **pair[k].i**, for k = 0,1,2,...N_p. A second pass through the data compares the sequential bits of the data to be embedded and sets the value of the host data element to the value **i** or **j**, according to the bit value to be embedded. If the bit-stream being embedded is random, the host data values **i** and **j** occur with equal frequency after the embedding process is completed.

Listing 4 shows code from the BMPEMBED program that performs the actual embedding. Listing 4 includes considerable housekeeping necessary to manipulate the data in the bit-stream and host data files. Line 1 and the lines following to line 12 allocate memory and initialize variables. The bit-steam data to be embedded are denoted the "data-image," and stored in the array **data_row[]**. The host data are denoted the "image-data," and are stored in the array **image_row[]**.

The index **li** is used in a loop beginning at line 12 to count the position byte in the data-image. The loop begins with **li** = -512 because a fixed amount of header information (512 bytes) is embedded before the data-image bits. Line 14 contains the test for loading **data_row[]** with the header information. Line 20 contains the test for loading **data_row[]** with bytes from the data-image file, **tape5**.

Line 30 in listing 4 starts a loop for the bits within a data-image byte. The variable **bitindex** = (0,1,2,...7) counts the bit position within the data-image byte **data_row[d_inrow]**, indexed by the variable **d_inrow**. The variable **lj** indexes the byte (pixel) in the image-data. Another variable, **inrow**, indexes the position within the image-data buffer **image_row[inrow]**. Line 32 tests for output of embedded data (a completed row of pixels) to the image-data file, and line 40 tests for completion of a pass through the image-data. In this example, one pass through the image-data is made for each of the pixel pairs, **pair[k]**, k = 0,1,2,....N_p.

In line 57, the pair index is incremented. A temporary pair data-structure named **pvalue** is used to hold the working values of the host data pixels being used for embedding. Line 60 provides for refreshing the image-data buffer, **image_row**. The embedding test is made at line 72. If the **image_row[inrow]** content equals the pair value representing a data-image bit of zero, no change is made in the image-data. However, if the bit-stream value is one, the image-data value is changed to equal **pvalue.j**. Line 84 treats the case for image-data values not equal to the embedding pair value. The bitindex variable is decremented, because the data-image bit is not yet accounted, and the image-data indices are incremented to examine the next host data value.

Listing 4

```
/*-------------------EMBEDDING CODE----------------------------*/
/*
              lj,          index over pixels in the image-data
              inrow,       index within the image-data row buffer
              nrow,        row number in the image-data
              li,          index over pixels in the data-image
              d_inrow, "   within the data-image row buffer
              k,           index within the PAIRS structure array
              maxval,  no. of bits embedded
              bitindex, bit position within the data-image byte
              byteplace,  position for read/write in tape6 file
*/
line 1        data_row = (unsigned char *)malloc((size_t)NCOLS);
              if(data_row==NULL)  {
                           pm_error("Data row data allocation failed!");
                           return(1);
                           }
```

[†] Increasing the complexity of the algorithm can randomize a repetitive pattern. For example, the host image pixel locations can be processed in random, rather than sequential order. Random pixel ordering can be seeded from the noise pattern in the original host.

Listing 4, continued

```
                        maxval = bit_place_index.maxval;
                        d_inrow = bit_place_index.d_inrow;
                        bit_place_index.li += d_inrow;
                        lj = (long)krow;
                        k  = 0;
                        nrow = -1;
                line 12 for(li=bit_place_index.li; li<length-NCOLS; li++)       {
                                bit_place_index.li = li;
line 14         if(li == -512L)         {               /* header information */
                        byteptr=(unsigned char *)&data_header;
                        for(d_inrow=0;d_inrow<sizeof(data_header);d_inrow++)
                data_row[d_inrow]=*(byteptr+d_inrow);
                        d_inrow = 0;
                        }
line 20         if((li >= 0L && (li % (long)NCOLS) == 0L) || reread != 0)         { /* next row of data-image  */
                        j = fgetpos(tape5,&tape5_pos);
                        j = fread(data_row, 1, (size_t)NCOLS, tape5);
                        if(!reread)  {
                                    for(i=0;i<j;i++) checksum += data_row[i];
                                    d_inrow = 0;
                                    bit_place_index.d_inrow = 0;
                                    }
                        reread = 0;             /* turn off flag for re-read on next Truecolor plane */
                        }
line 30         for (bitindex=bit_place_index.bitplace; bitindex<NO_BITS; bitindex++) {
                        bit_place_index.bitplace = bitindex;
line 32                 if((lj-krow) % (long)(BYTES_IN_ROW) == 0L)        {
                                if(nrow >= 0) {         /* write only after you read */
                                        inrow = fseek(tape6, byteplace, SEEK_SET);
                                        inrow = fwrite(image_row,1,(size_t)(BYTES_IN_ROW),tape6);
                                        byteplace += inrow;
                                        byteplace += pad;  /* skip pad bytes */
                                        inrow = krow;
                                        }
line 40                 if(lj/(long)OFFSET == 0L ||
                (lj+(BYTES_IN_ROW+pad))/(BYTES_IN_ROW+pad) > (unsigned \
        long)bh.rows)           {
                                        if(bailout())  { /* end of image-data--user termination */
                                                i = 1;
                                                goto QUIT;
                                                }
line 47                                 if(k==no_pairs)goto PLANE;  /* next plane of  image */
                                        lj = krow;    /* pick next pair and start over */
                                        pvalue.i=(unsigned int)pair[k].i;  /*zero*/
                                        pvalue.j =(unsigned int)pair[k].j; /* one */
                                        if(verbose && k>0) fprintf(stderr," %ld     ", pvalue.count);
                                        pvalue.count = 0L;
                                        if(verbose) fprintf(stderr,"\rEmbedding Pair %2d\
                                        (%3d,%3d)",\
                                                k, pvalue.i, pvalue.j);
                                        else fprintf(stderr,".");
                                        k++;
                                        byteplace = bh.pixeloffset;
                                        }
line 60                 nrow = (int)((lj-krow)/((long)BYTES_IN_ROW+pad));  /* read next row */
                        inrow = fseek(tape6, byteplace, SEEK_SET);
                        inrow = fread(image_row,1,(size_t)BYTES_IN_ROW,tape6);
                        inrow = krow;
                        } /* end new row (lj) test */
              /* Embed one byte */
                if(ip>=0 && pair[k-1].count==0)  { /* finished a pair */
                        lj +=OFFSET;
                        inrow += OFFSET;
                        bitindex--;
                        continue;
```

Listing 4, continued

```
                              }
line 72             if((int)image_row[inrow]==pvalue.i) {       /* find a zero value */
                              if(test((int)data_row[d_inrow],bitindex)) image_row[inrow]=(unsigned char)pvalue.j;
          maxval++;
                              if(pair[k-1].count==0)            {
                                        pm_error("\nPair count error!");
                                        i = 1;
                                        goto QUIT;
                                        }
                              pair[k-1].count--;
                                                  pvalue.count++;
                                                  if(bitindex==NO_BITS-1)bit_place_index.bitplace = 0;
                                                  }
                    line 84             else bitindex--;        /* haven't got this bit yet! */
                                        lj+=OFFSET;
                                        inrow+= OFFSET;
                                        } /* end of bitindex loop */
                              d_inrow++;
                              } /* end of li (data index) loop */
```

3.2 Embedding greyscale and Truecolor host data

Greyscale and Truecolor images contain intensity values at the pixel locations[†] . Truecolor images contain intensity, or brightness values for each of the three color planes, and greyscale images contain a single intensity value for each pixel. Testing the color and hue is not necessary for Truecolor images, because each color plane is an independent monochromatic greyscale image. The noise analysis is done separately for each color plane in a Truecolor image, and three independent keys are generated. For greyscale images, only one key is needed.

With the key known for the color plane, the embedding is accomplished as was done for pallet-format images. The key values specify pairs of pixel values that are interchanged as necessary to encode the bits of the embedded data. Modifying the pair values in a greyscale or Truecolor image can significantly alter the statistical properties of the image histogram. Therefore, the analysis to determine the key applies an additional constraint on the pair values requiring the overall frequency of occurrence of each of the values to be similar. For example, if the values 150 and 151 are selected as a candidate key pair, because they are within the noise limit (eq. 1), the pair must also meet the constraint on frequency of occurrence $|f(d_i) - f(d_j)| < \delta$. Ignoring this constraint affects the entropy of the combined image. For the example given, one key value, say 150, might occur many times in the image and the other, 151, only a few times. After embedding random bits, the two values will occur with about the same statistical frequency, and the combined image will respond differently to compression than before embedding. By enforcing the statistical frequency constraint for greyscale and Truecolor images, the entropy of the combined image differs little from the original host.

4. EXTRACTION AND SECURITY

4.1 General principles

Extracting embedded data is done by reversing the process used to encode the data-image bit-stream. An analysis of the embedded image-data set reveals the candidate pairs for extraction *for only the case where the individual statistical frequencies are unchanged by the embedding algorithm.* In the BMPEMBED example, the statistical frequencies are changed by the embedding process. The pair table used for embedding can be re-created by analysis of the original image-data, or by comparing the image containing embedded data with the original host. However, the pair table cannot generally be recovered from the embedded image-data.

[†] Greyscale pallet-format images contain pointers to the pallet entry. Because the pallet may not be ordered according to intensity value, the pixel values do not correlate generally with brightness in the image. Greyscale images contain intensity values, and no pallet is used.

The pairs selected for embedding constitute a "key" to extract the data-image from the image-data. The BMPEMBED algorithm sorts the pixel-pair values meeting the constraints of eqs. 1 and 2, in order of decreasing total number M_k to create a list of pairs. The BMPEMBED.EXE versions later than Ver. 1.33 randomize the ordering of the pair list. Thus, data embedding reduces the statistical properties of the host data to a list containing pairs of numerical values. The pair components take their values from the pixel information. Each pair element is a *unique* value found in the image data. Therefore, the ensemble of pair values (a byte string of length $2N_p$) is a key for a symmetric stream encryption algorithm[4]. If the bit-stream encrypts using the frequency of occurrence of the pair values in the host image, ciphertext could be produced similar to one-time pad encryption. Data embedding differs from encryption, because the host data retains its information content even though the modified host data contain the ciphertext generated from the bit-stream. Data embedding is therefore a means to produce *invisible ciphertext.*

4.2 Relation to Cryptography

Schneier[4] discusses subliminal messages in the context of digital signatures. Subliminal messages differ from embedded data because the former messages reside within the separate ciphertext of a digital signature. In host data containing embedded information, the cipertext resides entirely in the noise component of the data. Because it is not possible to separate the noise with certainty, the cipertext is invisible and undetectable without the key information.

The key bytes describe the noise characteristics of the host data (eqs. 1 and 2). After the embedding process is completed, the key bytes cannot be recovered exactly from the data, because embedding preseverves the statistical properties of the host data in ensemble, but not in detail. Clearly the key-pairs or their equivalent, the original host data, provide the embedding security.

The key is generated by applying the algorithm to original, unmodified host data. The key differs for each host data set, and in principle, access to the embedding algorithm is insufficient to re-create a key. However, data embedding as implemented in the BMPEMBED.EXE demonstration has somewhat less security, because many key bytes can be recovered with high probability by analyzing the image containing the embedded information.

The demonstration algorithm intentionally preserves most of the statistical properties of the host data by selecting as embedding pairs those having nearly identical frequency of occurrence. Thus, depending on the statistical properties of the bit-stream embedded into the host, the modified-host data contain values having about the same frequency of occurrence as the unmodified host data. Security for the BMPEMBED.EXE demonstration program remains high, however, because analysis of the embedded image data does not reveal the *ordering* of the key bytes for versions after Ver. 1.33. If an image is used once for embedding data, and it is unavailable to an attacker, a chosen plaintext attack is not possible.

If the key is destroyed after embedding, the process resembles one-time-pad encryption. Data embedding therefore possesses potentially high security. Obviously, many of the key byte-values can be found with near certainty by comparing the original and embedded data images. Because the key may use some bytes having infrequent occurrence, as few as one or two values in the entire host data, comparing original and embedded data hosts is never certain to reveal all the key pair values. Protecting the original host data is an important operational consideration. Key and host-data management determines the security of the embedded data..

Data embedding resembles most a stream cipher[4] operating in the counter mode, with a block algorithm. The internal state of the encryption algorithm is driven by the key values (the pixel pairs), according to the sequence of the pixel locations in the host image. The output function is the algorithm for manipulating the pixels in the host image. In the example given, pixels are interchanged as required to represent bit values. A more complex output function might use a table constructed from the pixel values to represent bit patterns. For example, eight equivalent pixel values could represent entries in an octal embedding scheme. Highly redundant host data favors a complicated output function.

It is possible to modify the data embedding internal state with feedback from the output function, in the manner of a cipher-feedback mode. For example, modifying the sequence of processing the host pixels, as suggested earlier, could be driven by the number of bits embedded in a row or a column, or by some other property of the manipulated pixels. A feedback modification can mitigate the error propagation inherent in the data embedding process.

In the BMPEMBED example, synchronization of the extraction process requires perfect fidelity of the image data. Modifying the image in a manner that changes pixel pair values unequally prevents data extraction[†] . In the example algorithm, a defect in a digital image prevents extraction of any information beyond the defect location. Modifying the embedding algorithm to embed blocks of data permits re-synchronizing the extraction scheme after processing corrupted pixels. One or more blocks may be lost owing to the defect, but the unmodified pixels extract correctly. We leave investigating the potential of cipher-feedback to maintain syncronization for future work.

4.3 Equivalent key length

The key size depends on the statistical character of the data, and its length is not fixed by the algorithm. The maximum size of the key is the set of combinations of all pixel values taken by pairs. Thus, if the pixel element value is 2^{ϑ}, the largest number of pairs is

$$N_p = 2^{\vartheta-1} \quad , \tag{3}$$

and the keylength in bits is

$$L = 2N_p\vartheta \quad . \tag{4}$$

Determining the key for embedded data by a direct plaintext attack requires 2^L combinations. For $\vartheta = 8$ (1-byte pixel size), $N_p = 128$, $L = 2048$. A plaintext attack, known or chosen, is lengthy owing to the large number of possible key values provided by highly redundant host data. Any of the various plaintext attacks are restricted severely by one-time use embedding and the generally inaccessible original host image. Unlike conventional cryptanalysis, an attack successful for one host data set gives no information useful for attacking any other example of embedded data.

With the pair table known, extracting embedded data consists of sequentially testing the pixel values to re-create the output bit-stream for the data-image. In the BMPEMBED algorithm, the pair table is inserted into the bottom row of the embedded image-data, where it is available for the extraction program. An option in the program permits removing the pair key and storing it in a separate file. Typically, the pair key ranges from a two to perhaps a hundred bytes in size. For the minimum case, L = 16, the security of data embedding results from the time-consuming process of data extraction as compared with standard encryption schemes, and from the difficulty of recognizing extracted data as plaintext. Indeed, for the most secure protocol, the embedded information is stochastic, because it is ciphertext data encrypted from an original plaintext.

We note, for the BMPEMBED algorithm, a ciphertext-only attack is *always* possible, because the first block of data embedded is a header containing a "magic number" signature. Embedding with an algorithm that omits or encrypts the header information precludes recognizing the extracted data. Therefore, we believe the embedded data are secure if the original image-data or the pair key are not known. If the key is destroyed after embedding, the method gives security from the protocol of a one-time pad encryption method.

Another way of protecting the pair table is to remove the key and encrypt it using public-key or other encryption. The encrypted key can be replaced in embedded image-data preventing extraction by unauthorized persons. The presence of an encrypted key gives no more information than the absence of a key, and even with a known plaintext attack against the magic number signature, the security is high owing to a long key length and a complicated extraction algorithm.

Embedding data into a host slightly changes the statistical frequency of occurrence of the values used for encoding the bit-stream. Compressed or encrypted bit-stream data serve admirably as pseudo-random bit-streams. Consequently, embedding pseudo-random data minimizes changes in the average frequency of occurrence of the values in

[†] Modifying pallet-format images to vary color balance, contrast, or brightness does not affect the embedded information, because the embedding is done with pointers rather than values. Even if the pallet entries become identical, for example by saturating their color components, the pointers to the pallet remain distinct and continue to convey the embedded information.

the embedding pairs. The existence of embedded data is not detected easily by analyzing the image-data. Indeed, we believe it may be impossible to detect embedded information with certainty.

4.4 Data embedding protocols

Unlike standard ciphers[4], data embedding combines the bit-stream into the covertext of separate image-data. The data-image bit-stream embedded into the host image represents plaintext although it may well be encrypted before it is embedded. The combination of the host and embedded data constitutes the ciphertext. Using stenography, the existence of ciphertext is not evident, because the content and meaning of the host information is preserved by the embedding process. We regard data embedding as distinct from encryption because *no obvious ciphertext is produced.*

Communication protocols for data embedding vary with the application. The most secure is a one-time data protocol. Alice and Bob share a common library of original host images. After embedding a message to Bob, Alice destroys the key and her copy of the original host image. Bob determines which of his original images to use when he receives the image from Alice, and he generates the key pair-values from his copy of the host image. Bob cannot extract the data with the generated key, because the pair values were randomized by Alice before they were used to embed the data. Thus, Bob must determine the proper sequence of the key pairs, perhaps by trying all the combinations, or Alice must communicate to Bob the random seed.

To simplify matters, Bob and Alice can agree to use values drawn from the original image noise component as the seed for randomizing the key pairs. A separate algorithm generates a random seed from the noise signature in the original host image. The image containing embedded data differs slightly from the original, and the seed cannot be generated from it by the algorithm. In this protocol, Alice and Bob use two public algorithms. Both work on the original, unmodified host image, and the security of the protocol depends on protecting this image.

Somewhat less secure is the keyed-image protocol, in which Alice generates the image and its embedding key. Alice creates a unique image, for example by scanning a film negative or digitizing a television signal. The key is created by analysis of the noise component, and Alice embeds the secret message. Alice encrypts the key with a standard encryption method, for example a public-key encryption protocol, and sends it to Bob. If Alice destroys the original image after using it to embed data, the security rests on the protection of the key sequence[†] .

A third variation permits Alice and Bob to use a public image for their host data. In a public-image protocol, Alice and Bob agree on a series of image-processing manipulations that affect the noise character of a host image. To communicate with Bob, Alice acquires a public image from a convenient data source, for example an image extracted from a Worldwide Web homepage, or from a commercial image library. Alice processes the image to create a new image that differs from the public copy, and that cannot be re-created easily, owing to the complexity of the secret processing algorithm. Possible processing methods follow from manipulations in the frequency space afforded by an image transform, and from non-linear warping.

Bob receives the image from Alice and obtains a copy of the proper public image. The public image is processed with the algorithm used by Alice, and the key is generated by analysis, permitting Bob to extract the data. Bob and Alice communicate only through the image that is exchanged, having agreed in advance on the source for their public images. When Alice and Bob communicate with a public image, security shifts to the protected algorithm used to process the public image. The advantage of this protocol is the reduction of the communication between Bob and Alice, and the reduced data storage of an algorithm, as compared to an image library[‡] .

Image data are used as examples in this discussion, but the method is not restricted to such data. Alice and Bob can use any sort of real-world data containing a noise component. Tables of floating point values, spread-sheets or databases, digital audio and video, etc. are materials suitable for data embedding protocols.

[†] The example algorithm determines the size of the key by a checksum value. The key pair-values can be padded to the maximum size with random bytes, and encrypted to ensure an attacker always faces with the *maximum* key length.

[‡] The processing algorithm can be script commands for public software.

5. DATA EMBEDDING FOR TWO-COLOR FACSIMILE (FAX) HOST DATA

5.1 Image characteristics

In application to greyscale and color-pallet host data, the noise component originates from uncertainty in the numerical values of the pixel data, or in the values of the colors in a pallet. This section examines facsimile (fax) images, using the same principles as above, with a view toward exploiting redundancy in the image. Fax images consist of black and white bitmap data, *i.e.* the data for image pixels are binary (0,1) values representing black or white, respectively. We denote a black and white bitmap image as a 2-color bitmap. The standard office fax machine combines the scanner and the digital hardware and software required to transmit the image via the telephone connection. Fax images are transmitted using a special modem protocol. Characteristics of the fax modem standards are given in many sources. One example, the User's Manual for the EXP modem[5], describes a fax/data modem designed for use in laptop computers. Fax transmissions made between computers are digital communications and the data are therefore suited to data embedding.

As used above, the embedding process is conducted in two parts: analysis and embedding. For a fax 2-color bitmap, image noise can add or subtract black pixels from the image. Thus, the *length of runs* of consecutive like-pixels varies. An example demonstrates the principle. The scanning process represents a black line in the source copy by a run of consecutive black pixels in the 2-color bitmap image. The number of pixels in the run is uncertain by at least ± 1 owing to the scanner resolution. Black and white images are redundant because run lengths differing by one convey the same information.

Applying data embedding to 2-color bitmap data therefore consists of analyzing the bitmap to determine the statistical frequency of occurrence of runs of consecutive pixels. The embedding process varies the length of runs by ± 1 pixel according to the content of the bit-stream in the data image. Host data for embedding are any 2-color bitpmap image suitable for fax transmission. Hardcopy can be scanned to generate the 2-color bitmap, or the image can be created by using fax printer-driver software in a computer. We use as an example, an image created by Microsoft® Word[6] and converted to a 2-color fax bitmap by BitFax software[7].

5.2 Run-length Analysis

Code to analyze the lengths of runs in a row of pixels in a 2-color bitmap image is given in Listing 5. The arguments to the routine **rowstats()** are a pointer to the pixel data in the row (one byte per pixel, containing zero or one in value), a pointer to an array of statistical frequencies, the number of columns (pixels) in the data row, and a flag for internal routine options. The options flag is the size of blocks, or packets, of the bit-stream to be embedded. The options flag is tested in line 9, and the routine **packet_col()** is used for a positive option flag. The **packet_col()** routine is given in listing 6, and its purpose is to ensure the first pixel in the data row starts in an even column number. The location of the first pixel in the row flags the start of a the data packets (*cf.* Section 5.3 for details).

Line 12 begins a loop to examine the runs of pixels in the data row. Runs between the defined values MINRUN and MAXRUN are examined by the loop. The **j**-loop and the test at line 15 locates a run of pixels, and sets the variable **k** to the index of the start of the run. The test at line 21 selects only blocks of pixels having length **i**, less than the length of the row. The loop in line 22 moves the pixel run to temporary storage in the **array block[]**.

Listing 5

```
line 1      int rowstats(unsigned char *data_row, long *histogram, int ncols, int packet_size) {
                    int i, j, k, l;          /* loop counters */
                    int runs=0;              /* return value */
                    int count;                          /* no. of pixels in the run */
                    char letter = 'A';       /* starting code for flagging runs in the row */
                    unsigned char block[MAXRUN+3];   /* a block containing the run being examined */
            /*  find first bit in the row & adjust as a packet flag */
```

Listing 5, continued

```
                    if(packet_size >=0) {
line 9                j = packet_col(data_row, packet_size, ncols);
                      }                                      if(ncols <=0) return(-1);
line 12             for(i=MINRUN;i<=MAXRUN;i+=2)  {  /* i is the runlength being searched */          k = 0;
                         for(j=1;j<ncols;j++)  {  /* NOTE: data_row[0] is assumed to be zero!! */
line 15                    if(data_row[j]==(unsigned char)ONE)  {
                               if(data_row[j-1]!=(unsigned char)ZERO) continue;
                               k = j;  /* a block start */
                               }
                           else continue;
                    /* find a block of data ending with a zero pixel */
line 21                  if(k+i+2 > ncols) break;
line 22                  for(l=k;l<k+i+3;l++) block[l-k] = data_row[l];
                         l = j;
line 24                  if(block[i+1] != (unsigned char)ZERO) goto NEXT;
                         if(block[i+2] >  (unsigned char)ONE ) goto NEXT;
                      /* examine block for pixel count */
                      count = 0;
line 28               for(l=0;l<i;l++)  {  /* all but last bit in block must = 1 */
                            if(block[l]==(unsigned char)ONE) count++;
                            }
line 31               count++;
                      l = j+1;
line 33               if(count == i+1)  {  /*  set all but last pixel in run to flag value */
                            if(histogram != NULL)histogram[i]++;
                            runs++;
line 36                     for(l=j;l<j+count-1;l++) data_row[l] = letter;
                            l++;
                            }

                NEXT:  for(j=l+1;j<NCOLS;j++) if(data_row[j]==(unsigned char)ZERO)break;
                    } /* end of row (j) loop */
                    letter++;
                    }  /* end of run (l) loop */

               return(runs);
               }
```

The two tests at lines 24 and 25 reject blocks having run lengths other than the one required by the current value of the **i**-loop. The embedding scheme selects blocks of length **i**, for embedding by adding a pixel to make the length i+1. Thus the run will contain either **i** or i+1 non-zero pixel values, according to the bit-stream of the embedded data. If the run store in the **block[]** array does not end in at least two zeroes, it is not acceptable as a run of length **i**, and the code branches to NEXT, to examine the next run found.

Line 28 begins a loop to count the number of pixels in the run[†] . The number found is incremented by one in line 31 to account for the pixel added to make the run length equal i+1. Line 33 contains a test ensuring that the run selected has the correct length. The **histogram[]** array for the run-length index **i** is incremented to tally the statistical frequency of the run. The data row bytes for the run are flagged by the loop in line 36, with a character to distinguish the runs located. The flagging technique permits the embedding code to identify easily the runs to be used for embedding the bit-stream. On exit from this routine, the data row bytes contain runs flagged with letter codes to indicate the usable pixel positions for embedding the bit-stream. The return value is the number of runs located in the data row. A return of zero indicates no runs within the defined limits MINRUN and MAXRUN were located.

5.3 Embedding two-color (FAX) host data

Fax modem protocols emphasize speed and they therefore do not include error-correction. Thus, fax transmissions are subject to drop-outs and to lost data, depending on the quality of the telephone line and the speed of the transmission. A successful embedding algorithm must account for corruption of some portion of the image data. In the application code we wrote for fax images, we used a variation of modem block-protocols to embed the data. Treating the 2-color image as a transmission medium, we embed the data in blocks, or packets, and provide for start and stop flags, and parity checks.

The start of a packet is signaled by an image row having a pixel in an even column. The packet ends when the number of bits contained in the block are extracted or, in the case of a corrupted packet, when a start-packet flag is located in a line. A checksum for parity and a packet sequence number are embedded with the data in a packet. Using this method, errors in the fax transmission result in the loss of some, but not all of the embedded data. The fax embedding method is always useful for ASCII text transmissions, but embedded binary files require an error-free fax transmission.

Listing 6 gives the simple algorithm for initializing the 2-color bitmap lines to flag the start of the packets. Each row in the 2-color image contains a non-zero value beginning in an even column (packet start) or in an odd column (packet continuation).

Listing 6

```
                int packet_col(unsigned char *data_row, int packet_size, int ncols) {
                        int i;
                /* find first bit in the row & adjust as a packet flag */
line 4          for(i=1;i<ncols;i++)  {
line 6                  if(data_row[i]==(unsigned char)ZERO)    {
                          if(packet_size<0) break;
                          if(packet_size>0)  {          /* first bit set to an even column */
                                if(i%2 == 0)break;
                                data_row[i] = (unsigned char)ONE;
                                }
line 11                   else {  /* first bit set to an odd column */
                                if(i%2 != 0)break;
                                data_row[i] = (unsigned char)ONE;
                                }
```

[†] Owing to the convention for fax-format images, the runs examined are *whitespace* runs instead of black, image pixel runs.

Listing 6, continued

```
                              }
                          }
line 17        if(i==ncols)return(-1);          /* no black pixels in the row */
               if(packet_size>=0) return(i);    /* index of the first black pixel */
               if(i%2) return(1);               /* if(packet_size==-1) return odd */
               else      return(0);             /*                       return even */
               }
```

Line 4 in listing 6 starts a loop over the number of pixels in a data row. In fax images, a zero pixel value indicates black space and a 1-value indicates white space. Line 5 locates the first black space in the data for the row. If the variable **packet_size** is positive, the column index is tested to be even and the pixel is forced to white space. If the **packet_size** variable is negative, the routine returns an indicator of the data row flag. If **packet_size** is greater than zero, the first data row element is flagged as white space. Line 11 treats the case for **packet_size** = 0, indicating a continuation row. For a continuation row, the first data row element is forced to black space.

Data embedding is accomplished by code that examines the contents of the data row after it has been analyzed and flagged with letter-codes to indicate the run lengths. The code fragment in listing 7 shows the embedding process for 2-color bitmaps. Lines 1 through 49 complete a loop (not shown) over the index **lj**, in the 2-color bitmap image. Lines 1 through 26 contain code to read one line of pixels from the 2-color bitmap. The row number in the image is stored in the variable **nrow**, in line 1. Data are read by bytes from the image file in the loop beginning at line 12. The bits are decoded and expanded into the **image_row[]** array. The **image_row[]** array contains the pixel values stored as one value (0 or 1) per byte.

Line 28 uses the **packet_col()** routine to return the packet-index for the row. If the return value **j** in line 28 is 0, the row is a packet-start row, and if **j** is 1 the row is a continuation row. Line 29 uses the **rowstats()** routine to assign run-length letter flags to the pixels in the row buffer. The return value, **i**, gives the number of runs located in the image row. Consistency tests are made at lines 31, 37, and 41. The index **kp** gives the line number within a data packet. If **kp** is 0, the line must be a packet-start index, and if **kp** > 0, the line must be a continuation line. Line 49 completes the code segment to read and pre-process a line of 2-color image data.

The data-structure array **pair[]** contains the run length for (i), the augmented run length (i+1), and the total number of runs in the 2-color bitmap image. The index **k** in the loop starting at line 51 is the index for the run length being embedded. The index **inrow** counts pixels within the image row buffer, and the variable **bitindex** is the bit-position index in the bit-stream byte.

Line 57 sets the value of the run-length letter-flag in the variable **testltr**. The value of an image pixel is tested against the letter flag in line 58. If the test letter-flag is located, line 60 advances the index in the row to the end of the pixel run being used for embedding. The test function in line 62 checks the bit value for the bit index in the bit-stream packet byte. If the value is one, the last pixel in the run is set to one. Otherwise, the last pixel in the run is set to 0. The number of bits embedded is recorded in the variable **maxval**.

Listing 7

```
line 1         READLINE:        nrow = (int)(lj/((long)bh.cols));   /* read data from next row */
                                if(verbose) {
                                          if(nrow==0)fprintf(stderr,"\n");
                                           fprintf(stderr,"\rrow %4d", nrow);
                                           }
                                else motion(stderr);
```

Listing 7, continued

```
            bit_count = 0;
            image_row[0] = 0;    /* row buffer always starts with a zero */
            if(verbose==2 && nrow <=61)fprintf(tape9,"\nnrow byteplace %d %ld", nrow,byteplace);
            inrow = fseek(tape6, byteplace, SEEK_SET);
            writeplace = byteplace;
line 12     for(j = 1; j < (int)bh.cols+1; j++) {
                    int pix;
                    if(bit_count <= 0) {    /* need another byte */
                            bit_count = 8;
                            bit_store = pbm_getrawbyte(tape6);
                            byteplace++;
                            }
                    bit_count -= bh.bitsperpixel;
                    pix = ( bit_store >> bit_count ) & mask;
                    image_row[j] = (unsigned char)pix;
                    #ifdef INSERT_KEY
                    /* key row set to zero to hold key pairs */
                    if(nrow == KEYLINE)image_row[j] = (unsigned char)ZERO;
                    #endif
line 26             } /* cols */
            byteplace += pad;
line 28     j = packet_col(image_row,-1,(int)bh.cols);
            i = rowstats(image_row,NULL,(int)bh.cols+1,-1);    /* flag the embedding pixels */
            if(verbose==2) fprintf(tape9,"\n nrow,i,j: %d %d %d", nrow,i,j);
line 31     if(j<0 || i==0)         { /* a row of white pixels or no pixels for embedding  */
                    if(nrow+1<(int)bh.rows)  {
                            lj += bh.cols;
                            goto READLINE;
                            }
                    }
line 37     if(j==1 && kp==0)  {
                    fprintf(stderr,"\nPacket start-index error, packet %d", packet_no-1);
                    goto QUIT;
                    }
line 41     if(j==0 && kp > 0)  {
                    fprintf(stderr,"\nContinuation packet-index error, packet %d", packet_no-1);
                    goto QUIT;
                    }
            inrow = 1;
            if(kp==0 && verbose==2)
```

Listing 7, continued

```
                                   fprintf(tape9,"\nPacket start-row %d,  bits found %d",nrow,i);
                                 kp++;
line 49                          } /* end new row (ij) test */
        /* Embed one byte, use all pairs for each row */
line 51            for(k=0;k<no_pairs;k++) {
                      if(pair[k].count<0) {
                              pm_error("\nPair count error!");
                              i = 1;
                              goto QUIT;
                              }
line 57               testltr = (unsigned char)(letter+(unsigned char)pair[k].i/2 -1);        /* flag letter */
                      if(image_row[inrow]==testltr)          {             /* find a flagged run */
                              if(verbose==2 && nrow==60) fprintf(tape9,"inrow %d", inrow);
line 60                       inrow += (unsigned int)pair[k].j;
                              ij += pair[k].j;
line 62                       if(test((int)packet[inpacket_row],bitindex)) image_row[inrow-1]=1;
                              else image_row[inrow-1]=0;
                              }
```

Setting the value of the pixel trailing a run implements the embedding in the 2-color bitmap images by introducing noise corresponding to the pseudo-random bit-stream in the packet data. The letter-flag values written into the row buffer by the call to **rowstats()** in listing 7 are reset to unit (1) value before the **image_row[]** array data are packed and written back to the .BMP format file. This code is not shown in listing 7.

5.4 Data extraction from two-color (FAX) host data

Data embedded into a 2-color bitmap fax image can be extracted easily, *if the transmission of the fax is received by a computer.*. The image data are stored by the receiving computer in a file format (usually a fax-compressed format, see Levine[2] for details) permitting processing to extract the embedding data. Fax data sent to a standard office machine *are not* amenable to data extraction, because the printed image is generally not of sufficient quality to scan for recovery of the embedded data.

5.5 Two-color embedding keys

The key for 2-color image embedding can be recovered by analyzing the embedded image, because the run lengths are not changed from the original (i, i+1) values. The order in which the values are used depends on the frequency of occurrence in the image. As in the example for pallet-color images, the values of the pairs used for embedding is inserted into the fax to provide a key for extraction. However, the key is not strictly required, because in principle, knowledge of the defined values MINRUN and MAXRUN permits re-calculating the run-length statistics from the received image. In more complicated algorithms, the key is required. Thus, while somewhat less secure than for pallet-color images, the 2-color bitmap algorithm can be implemented still as a one-time pad cryptographic protocol.

6. DIGITAL WATERMARKING

6.1 Watermarking concept

Images presented in digital form are readily copied and edited for use in publications. Common methods to distribute images rely on reduced-resolution samples, or encryption. A means of impressing a *removable* digital watermark facilitates distributing image products. Images are distributed with overlay patterns that prevent their use, but require these require obtaining a separate copy of the image for use without the pattern overlay.

Data embedding techniques provide a means of distributing images with removable watermarks. The algorithm given here is implemented in BMPEMBED.EXE Ver. 1.50 and later. An overlay mask, or pattern, is defined in a data file. The watermarking of the host image follows the scalable mask data. Briefly, each pixel in the host image is tested to determine if it is located within the scaled mask. If the pixel lies outside the watermark mask data, no action is taken. If the pixel is within the watermark, its value is written to a data file and the host file is modified by replacing the pixel with new information. After completing the watermarking process, the data file of the pixels removed from the watermarking region is embedded *into* the watermarked image.

Removing the watermark requires the embedding key to extract the pixel data file. The overlay mask, or pattern file, or a pattern algorithm is also required. The process described above is reversed to remove the watermark. A host pixel found to be within the watermark pattern is replaced with the original value taken from the pixel data file.

6.2 Watermarking algorithm description

The code fragment in Listing 8 shows the code used to impress the watermark on the host image. Prior to executing this fragment from the routine **waterBMP()**, the host image has been analyzed for data embedding potential. The key pairs have been selected and the header information is available.

In line 1 of Listing 8, the index for color plane, **ip**, is incremented if the host image is in Truecolor format. Line 4 initializes the key-pair data for Truecolor images. If the image is in pallet color format, the pair values are initialized by the analysis process executed in previous routines. Line 10 begins the watermarking code. The host data are located in the file pointed to by the handle "**in**," and the watermarked data will be written to the file pointed to by the handle "**tape6**." The pixel data begin at the offset location stored in the bitmap format header structure member **bh.pixeloffset**.

Lines 15 through 51 fall within a loop over the index **nrow**, the row number in the host image. The watermarking process is performed by code that operates on the pixels within an individual row. The file is positioned in line 18, and a row of pixels is read (line 19). The pixel byte values for the row are available in the memory array **image_row**. The index **i** interspersed in code within the loops will be discussed below. The loop using the **j** index, beginning in line 22 steps the calculation across the pixels in a row. The variable **krow** provides an offset in the loop for Truecolor images. Truecolor images interleave the RGB color plane values and **krow** offsets the **j** index appropriately.

Listing 8

```
line 1          if(bh.bitsperpixel == TRUECOLOR) {
                   if(ip>0)fp = tape6;
                   no_pairs = tc_pair[ip].no_pairs;
line 4             for(i=0;i<no_pairs;i++) {
                          pair[i].i = (color_pair[ip]+i)->i;
                          pair[i].j = (color_pair[ip]+i)->j;
                          pair[i].count = (color_pair[ip]+i)->count;
                          }
                   }
line 10         rewind(in);
                rewind(tape6);
                byteplace = bh.pixeloffset;
                i = 0;
                if(verbose)fprintf(stderr,"\n");
line 15         for(nrow=0;nrow<(int)bh.rows;nrow++) {
                   if(verbose)fprintf(stderr,"row %4d\r",nrow);
```

Listing 8, continued

```
                else if(nrow%10 == 0)fprintf(stderr,".");
                j = fseek(fp,byteplace,SEEK_SET);
                j = fread((void*)image_row,1,(size_t)(BYTES_IN_ROW),fp);
                i++;
line 21         if(i==no_pairs)i = 0;
line 22         for(j=krow;j<(int)(BYTES_IN_ROW);j+=(int)OFFSET)   {
                    if(nrow==0) image_row[j] = 0;       /* zero the last row for pair key */
                    else {
line 25               k = patmask(nrow,j);
line 26               if(k)  {
line 27                   data_row[d_inrow] = image_row[j];
                          checksum += (unsigned long)data_row[d_inrow];
                          d_inrow++;
                          maxval++;
                          if(d_inrow == NCOLS)  {
                             k = fwrite((void *)data_row, 1,(size_t)(NCOLS),tape5);
                             d_inrow = 0;
                             }
line 35                   beta = rand();   /* Watermark the pixel with a pair value */
                          if(beta > RAND_MAX/2) image_row[j] = (unsigned char)pair[i].i;
                          else                  image_row[j] = (unsigned char)pair[i].j;
                          }
                      }
                   }  /* j loop */
line 41         j = fseek(tape6,byteplace,SEEK_SET);
                j = fwrite((void *)image_row,1,(size_t)(BYTES_IN_ROW),tape6);
                byteplace += j;
                pad = 0;
                while((byteplace-bh.pixeloffset)&3 )  {    /* read & write pad bytes */
                     (void)pbm_getrawbyte(fp);
                     if(fp!=tape6)putc(0,tape6);
                     byteplace++;
                     pad++;
                     }
                }  /* nrow loop */
line 52       if(ip < 2)goto START;             /* loop for TrueColor image planes */
```

Line 26 performs the test to determine if the pixel is within the watermark pattern. The return value **k** directs execution to the next pixel if the current one is outside the watermark, or into code that replaces the pixel with new information if the current one is inside the watermark. Pixels within the watermark mask are saved in the memory array

data_row, in line 27. A running checksum is used to validate the subsequent removal of the watermark. The memory buffer **data_row** holds NCOLS pixel values and it is periodically flushed to the **tape5** file.

Once a pixel is saved, it must be replaced in order to impress the watermark. Many schemes are possible, the simplest being a replacement with white or black. In Listing 8, we replace the pixel with a value chosen randomly from the key-pair table. Line 35 selects a random number **beta,** and lines 36 and 37 replace the host pixel with one of the two possible values from the key-pair indexed by the variable **i.** As a result, the watermark in the image contains values already occurring in the image and known to be useable for embedding data. This watermarking algorithm is less obvious in the final image than a simple contrasting logo, but the space available for embedding data is increased. In line 41, the output file is repositioned and the modified pixel data written.

Line 52 completes a pass through the image. For pallet format files, the watermarking process is completed, and for Truecolor files the color plane index **ip** is incremented and the process repeated. After the watermark is impressed, the data file written to **tape5** is embedded into the image and the process is complete.

6.3 Watermark removal

Extracting the watermark reverses the process described for Listing 8, starting with a data file *extracted* from the watermarked image. The clear advantage of data embedding in the watermarking process is the ability to include the information required to remove the watermark in the noise component of the image. The present application embeds only the pixels needed to reconstruct the image. The watermark mask, or pattern, must be available separately and it must agree exactly with the one used to mark the image. Alternatively, a scalable watermark algorithm can define the pattern.

6.4 Pattern thinning

The watermarking algorithm in the example permits a single mask pattern to be scaled as needed to fit a variety of host images. The watermark mask is scaled by the host image size, and by the number of watermark pixels permitted. Size scaling is not complicated and will not be discussed in detail. However, scaling the number of watermark pixels is more subtle, and is discussed with reference to the **thinpat()** routine in Listing 9. This code is used to reduce the number of pixels in the mask pattern to a number that can be embedded in the host image. Some images, particularly the pallet format ones, have less embedding space than the number of pixels in the user-defined mask file. Therefore, fewer watermarking pixels must be used. The algorithm in Listing 9 reduces the number in a dynamic way that is reproduced exactly when the pattern is used again to remove the watermark.

Listing 9

```
/*********************************************************************
 *        THINPAT:  Routine to 'thin' a pattern mask to reduce the no. of pixels
 *                  in the mask to an amount that can be embedded into the
 *                  host image.
 *        INPUT:    mask_factor,  global double, a factor for scaling (reducing) the mask
 *                  pattern.
 *        RETURN:   *new,  FILE pointer to the new pattern file.
 *                         NULL pointer if an error occurred.
 *        AUTHOR:   M. T. Sandford II, March 21, 1995
 *                  Copyright (c) Univ. of California, All Rights Reserved.
 *********************************************************************/
FILE * thinpat(FILE *tape10)   {
   unsigned long pixel_index;
   unsigned long ij;
   unsigned long no_pat_pixels;
   unsigned char pixel_value;
```

Listing 9, continued

```
                unsigned char xtmp;
                unsigned short start=SEED_VALUE;   /* starting seed for srand() */
                int i,j;
                float x_frac,y_frac;
                FILE *new10;
                /*  copy header from .HSI raw format file to a temporary mask file */
                fprintf(stderr,"\nThinning watermark pattern...");
line 1          rewind(tape10);
                thinfile = _tempnam(".","thin");
                strcat(thinfile,".raw");
line 4          new10 = fopen(thinfile,"w+b");
line 5       for(i=0;i<water_file_pos;i++)     {
                     j = fread(&pixel_value,1,1,tape10);
                     j = fwrite(&pixel_value,1,1,new10);
                     }
                /* read & randomly thin the pattern mask, writing to file new10 */
line 10         srand(start);  /* seed the random no. generator */
line 11         x_frac = (float)(1./ mask_factor);          /* percentage of mask to retain */
                pixel_index = hsi_header.width;
                pixel_index *= hsi_header.height;
                no_pat_pixels = 0L;
line 15         for(ij=0;ij<pixel_index;ij++)  {
                      i = fread(&pixel_value,1,1,tape10);
                      if(i != 1)  {
                        fprintf(stderr,"\nTHINPAT.C:  I/O error");
                        }
                      /* rejection sampling to scale pattern size to embedding space */
                      xtmp = pixel_value;
line 22               if(!pixel_value)  {
                         y_frac = (float)rand()/(float)RAND_MAX;
                         if(y_frac > x_frac)xtmp = 0x01;  /* reject the mask pixel */
                         else no_pat_pixels++;
                         }
                     j = fwrite(&xtmp,1,1,new10);
                     }
             fclose(tape10);
             tape10 = NULL;
             fclose(new10);
             fprintf(stderr,"%ld pattern bytes",no_pat_pixels);
             new10 = fopen(thinfile,"r+b");
```

Listing 9, continued

```
        return new10;
        }
```

The routine takes as argument the file handle for the watermark mask. The routine thins this pattern and creates a new mask, or pattern file. The handle to the new file is returned to the calling program. Lines 1-4 define the data files used. The loop starting at line 5 copies the header information from the user-supplied pattern file to the new file. The watermark file is in a binary format defined by Handmade Software, Inc.[8]

Line 10 seeds the random number generator with a starting value stored in a global variable. The seed value used for the random number generator is stored in the header of the data file embedded into the watermarked file, ensuring that *the identical random number sequence is generated when the watermark is removed.*

In line 11, the fraction of the mask pixels to retain is calculated using the global variable **mask_factor**. For example, if **mask_factor** = 3.0, the user-supplied mask has been determined to be three times larger than the number of pixels that can be embedded into the image. The fraction of pattern pixels to retain is therefore 1/3.

Line 15 starts a loop over the total number of pixels in the pixel mask file. The mask file contains one pixel value per byte, and a non-zero byte indicates white space. A zero pixel value indicates information within the watermarking pattern. Lines 22 through 26 consititute code for rejection sampling the pixels in the mask file. Statistically, the fraction **x_frac** of the user-defined pixels is retained. The new file contains a pattern that resembles the original watermark, thinned by missing pixels.

Removing the watermark requires that the mask file be thinned identically, as in the marking process. The watermark is therefore secured additionally in images having relatively small embedding space. Typically, in pallet format images, about 5% of the image space is usable for embedding and this permits a modest logo or text string to be used for watermarking. In Truecolor images, the embedding space is typically so large that considerable space is available and no thinning is required.

7. COMPRESSION EMBEDDING

7.1 Lossy Compression

When it is necessary to transmit large amounts of data, innovative methods to minimize the communication time are required. Transmissions of digital television, for example, use complicated data compression methods to accomplish this minimization. A class of these methods is termed "lossy compression." The class is termed 'lossy' because the compression methods reduce slightly the quality of the original data. Multi-media computing applications use lossy compression of image and audio data to improve performance and reduce data storage requirements. Most lossy compression methods depend on a transform from image pixel values to coefficients of a series expansion.

Redundancy and uncertainty are intrinsic to lossy compression methods. Two examples of lossy compression are the Joint Photographic Experts Group (JPEG) standard, and the Wavelet Scalar Quantization (WSQ) algorithm that has been adopted by the Federal Bureau of Investigation for the electronic interchange of digital fingerprint information. The JPEG algorithm is based on the Discrete Cosine Transform (DCT) representation of the host data. The WSQ method is based on a representation of the host data in terms of wavelet functions. In both methods, the host data representation exists in an intermediate stage as a sequence of integer values referred to as 'indices.'

Data embedding principles appear at first incompatible with lossy compression, because the loss modifies the noise component and degrades slightly the fidelity of the original host data. This loss in fidelity destroys any information which has been embedded into the noise component of the host data. However, it is possible to define an embedding algorithm in the transform of the image, in the space of coefficients.

At the intermediate stage, loss of fidelity has occurred because the transform coefficients representing the data are quantized to a finite number of integer representations. Redundancy occurs in lossy compression methods because an integer value is uncertain by typically one unit in value. Thus, although adjacent integers occur many times in the compression sequence of indices, they often contribute equally to the reconstructed data. Uncertainty occurs in the integer representation because the uncertainty in the original host data carries through to its transform representation. The integer representation values are individually uncertain by at least ±1 unit of value, and this redundancy can be exploited in fashion similar to that used for embedding into black and white images.

The JPEG algorithm is used to demonstrate compression embedding. The method is given in an article by Gregory K. Wallace[9]. The JPEG algorithm is used primarily for compressing digital images. A somewhat less technical, more leisurely introduction to JPEG can be found in the book by Mark Nelson[10]. Pennebaker and Mitchell[11] published a detailed textbook about JPEG. The JPEG format is represented by ISO standards DIS 10918-1 and DIS 10918-2. The Independent JPEG-Group's C-language source code is available electronically from ftp.uu.net (Internet address 137.39.1.9 or 192.48.96.9). The most recent released version can always be found there in directory graphics/jpeg.

The WSQ method as applied to compressing digital fingerprint images was given by Bradley and Brislawn[12], and by Bradley, Brislawn, and Hopper[13]. Documentation for WSQ compression is available through Jonathan N. Bradley, Los Alamos National Laboratory, P. O. Box 1663, MS-B265, Los Alamos, NM 87545, and electronically from the Internet FTP site "ftp.c3.lanl.gov" in directory /pub/WSQ.

7.2 Compression embedding concept

The actual embedding of the auxiliary data into the compressed representation of integer indices is a three-part process. First the indices, representing the transform coefficients, are examined to identify pairs of the integer indices having values that occur with approximately the same statistical frequency, and that differ in value by only one unit. Second, the order of the integer indices pair values is randomized to generate a unique key sequence that cannot be duplicated by an unauthorized person. Third, the pairs of indices identified in the compressed integer representation are used to re-order the indices in the compressed representation in accordance with the bit values in the sequence of auxiliary data bits. The key sequence is optionally appended to the compressed data file.

Extracting embedded data inverts this process. The key sequence of pairs of index values is recovered from the compressed data file, or it is supplied as information separate from the compressed data. The key specifies the pair-values of indices differing by one unit in value. With the pair values known, the extraction consists of recreating the auxiliary data according to the sequence of occurrence of the indices in the compressed representation. The key data are used first to extract header information. The header information specifies the length and the file name of the auxiliary data, and serves to validate the key. If the compressed file contains no embedded information, or if the incorrect key sequence is used, the header information will not extract correctly. However, successful extraction exactly recreates the auxiliary data in an output file.

7.3 Analysis of Compression Indices

. The JPEG method compresses the host image in pixel blocks specified to the algorithm at the time the indices are calculated. The WSQ method compresses the host image by passing it through a series of multirate filters. In both the JPEG and WSQ algorithms, the image host data exist in an intermediate stage as a sequence of integer (16-bit) indices. The indices represent an image originally presented in a standard digital format.

The characteristic of lossy compression that makes compression embedding possible is redundancy owing to uncertainty in the index values. Each integer index occurs typically many times in the compressed representation, and each index is uncertain in value due to uncertainty in the host data. The analysis algorithm creates a histogram of the integer indices in the compressed representation. This histogram shows the probability density of the integer values in the representation, and plots the number of times a particular value occurs versus the value. For JPEG compression, values in the range ±1024 are sufficient to demonstrate the method, and for WSQ compression we use values in the range ±4096. A particular distribution of values depends on the image content, but both compression methods concentrate the values in a pattern symmetrical about 0.

Figure 1 is a reproduction of the greyscale image example. Compression embedding works equally well with color images, because the chroma, hue, and luminance are separated by a different algorithm, before the expansion to integer indices. Details of color image handling differ among the respective lossy compression algorithms. Thus, the grey scale image example in Fig. 1 does not represent a limit to the application of the embedding method. The grey scale example image is reproduced in Fig. 1 at reduced scale and quality to facilitate printing. The reproduction in Fig. 1 demonstrates only the nature of the image example, and the figure does not convey a representation of the image quality appropriate to evaluate the performance of the data embedding example. Images expanded from compression representations containing embedded data are visually identical to those expanded from unmodified compressed images.

Histograms of the indices for the image example in Figure 1 are shown in Figures 2 and 3, for JPEG and WSQ compression respectively. Fig. 2 shows the histogram for the JPEG compression representation, for the image sample shown in Fig. 1, and compression ratio about 12:1. The file size for the JPEG version of Fig. 1 is 42953 bytes. Analysis of this image according to the algorithm used for this example identifies 50 pairs of values in the histogram, totalling 3573 bytes of embedding space, slightly less than 10%.

Figure 3. shows the histogram for the WSQ coefficient representation of the image sample shown in Fig. 1, and compression ratio about 20:1. Analysis of these coefficients identifies 37 pairs of values in the histogram, totalling 471 bytes of embedding space. The WSQ histogram contains fewer coefficients than produced by the JPEG method owing to the larger compression ratio, but their redundancy nevertheless permits embedding some information into the compressed representation.

Compression embedding involves the rearrangement of certain values in a lossy compression representation in order to encode the values of the extra data which is to be added. For the purpose of illustration, consider the compression representation consists of a continuous sequence of integer values or indices. Further assume that any intermediate index value is uncertain by ±1 unit in value. The frequency of occurrence or histogram value of a certain index i is f(i).

Two values i and j in the table of indices are candidates as embedding pairs if:

$$|i - j| = 1. \tag{5}$$

For compression embedding, j = i+1. Index values meeting the criterion of Eq. 5, and occurring also in the representation with $f(i) - f(j) < \delta$, where f(i) and f(j) are the probability of occurrence of adjacent intermediate index values, and δ is the tolerance imposed for statistical equality, are candidates for embedding use. The values i and j meeting this constraint constitute a *pair* of index values p_k. There are $k=0,1,2,...N_p$ such pairs in the compression representation, giving a total number M of embedding bits:

$$M = \Sigma\, i(\ell) + \Sigma\, j(\ell) \tag{6}$$

The summations over ℓ of i and j run over a limited range of the intermediate indices. In the example given here, the summation limits are specified at ±1024.

In listing 10, the loop beginning at line 3 processes the histogram table to identify pairs of indices meeting the specifications above. The table values are stored in the variable **hist_table**, having H_TABLE_SIZE entries. Figures 2 and 3 show the histogram data for sample JPEG and WSQ compression representations. The code in line 8 compares the absolute difference of the frequency of occurrence f(i) and f(j) with the average value of the two occurrence frequencies. Pairs differing by less than the average value are accepted in this example. This simple selection scheme prevents an artificially large modification to the indices in the compression representation. For example, if f(i)=1000 and f(j)=500, the absolute difference is 500 and the average is 750. This pair will be rejected as an embedding candidate. However, if f(i)=1000 and f(j)=750, the absolute difference and the average are 250 and 875, respectively, and this pair will be accepted. This, or a similar scheme for selecting pairs for approximate equality of their component's frequency of occurrence minimizes perceptible differences in the image expanded from the compressed representation containing embedded data.

Listing 10

```
                /*          Process histogram to select embedding pairs */
                j = 0;
line 3          for(i=0;i<H_TABLE_SIZE;i++)          {
                        lsum = hist_table[i]+hist_table[i+1];
                        lavg = lsum/2L;
                        ldiff = hist_table[i]-hist_table[i+1];
                        ldiff = labs(ldiff);
line 8                  if(ldiff < lavg) {  /* Difference less than avg. for pair */
                                pair[j].i = i+(int)minval;
                                pair[j].j = i+(int)minval+1;                          pair[j].count = (unsigned long)lsum;
                                j++;
                                i+=2;
                                if(j==MAXPAIRS)break;
                                }
                        }
                lsum = 0L;
line 18         no_pairs = duplicate(j,pair);
                for (i=0;i<no_pairs;i++) {
                  if(pair[i].i==pair[i].j)break;
                  printf("\npair[%3d] %4d %4d %5ld", i,\
                                          pair[i].i, pair[i].j, pair[i].count);
                  lsum += pair[i].count;
                }
                no_pairs = i;
                printf("\n%d pairs located.  %ld total embedding bits", no_pairs, lsum);
```

The pairs selected from the histogram are stored in the data structure array element **pair[j]** in lines 9, 10, and 11. In the example in listing 10, the data structure permits MAXPAIRS pairs to be selected. The structure element **pair[j].count** contains the total number of occurrences of the (i,j) values in the histogram table. Line 18 uses the routine **duplicate()** to remove duplications from the pair table. Code starting in line 19 calculates the total number of pairs, **no_pairs**, and **lsum**, the total number of bits that can be embedded into the compression indices. M, in Eq. 6 defines the calculation performed in the loop starting at line 19.

The embedding process ignores completely the contribution the individual index values make to the compression representation. In JPEG compression, the values represent the coefficients in a discrete cosine transform performed over pixels in a square block of the image data. Usually, 8 x 8 pixel blocks are used, but the details of the transform and the tiling of the image data are irrelevant for embedding purposes. In WSQ compression, the indices are determined by quantizing the discrete wavelet transform coefficients which are calculated by repeated applications of a multirate filter bank. Again, details of the wavelet calculations and the sampling size are ignored in the selection and use of the embedding pairs.

Depending on the details of the selection algorithm, the index pairs found can include generally redundant values. The same index value i, is found perhaps in several different pair combinations. Because multiple pairs cannot contain the same index entry, due to each pair combination of index values having to be unique, it is necessary to eliminate some pairs. The number of pairs located by applying the criterion of Eq. 5 is stored in the variable **j**, in line 18.

The security of the embedded data is increased significantly if the pair values are arranged into a random order. Randomizing the order of the pair values is an important part of data embedding, and we illustrate it in Listing 11. Randomizing is accomplished by rearranging the pair values according to a randomly ordered data structure. The structure named **index_pts** contains elements **index_pts[k].i**, k=0,1,2,..., **no_pairs**; and **index_pts[k].gamma**, $\gamma_1,\ldots,\gamma_k,\ldots\gamma_{no\_pairs}$, where the γ_k values are uniformly random on (0,1). The standard library routine **qsort()** is used to sort the data structure **index_pts[]**. Putting the random element values into ascending order randomizes the index element of the structure. The random index values are used with the pair structure elements calculated and sorted as indicated above, to re-order the table to give random pair ordering.

Listing 11

```
/*************************************************************************
 * JUMBLE.C:  Routine to jumble a table of indices using pseudo-random
 *               numbers seeded from the PC clock
 * INPUT:     index, pointer to a table of jumbled integers
 *            npts,  integer no. of entries in the index table
 * RETURN:    nothing
 * OUTPUT:    returns a jumbled table of integers
 * AUTHOR:    M. T. Sandford II, 5 Oct. 1994, following the method
 *               preferred by T. Handel
 ************************************************************************/
void jumble(int *index, int npts) {
int i;
float fi;
struct POINTS {
   float gamma;
   int i;
   } *index_pts;
index_pts = malloc(sizeof(struct POINTS)*nindex_pts);
for (i=0;i<nindex_pts;i++) {
   index_pts[i].i = i;
   index_pts[i].gamma = (float)rand()/(float)RAND_MAX;
   }
qsort( (void *)index_pts, (size_t)nindex_pts, sizeof(struct POINTS),
                                       index_compare);
for(i=0;i<nindex_pts;i++) {
            index[i] = index_pts[i].i;
/*     printf("\ni,index[i] %d %d", i, index[i]); */
            }
if(index[0]==0) {
   fi = (float)rand()/(float)RAND_MAX;
   fi *= (float)(nindex_pts-1)+1.0f;
   index[0] = index[(int)fi];
   index[(int)fi] = 0;
   }
free(index_pts);
}
```

Listing 11, continued

```
/*  comparison routine for sorting points structure values  */
int index_compare(const void *p1, const void *p2) {
   if((float *)p1 > (float *)p2)return(1);
   if((float *)p1== (float *)p2)return(0);
   else return(-1);
   }
```

7.4 Compression embedding

The actual embedding of auxiliary data into a compression representation consists of rearranging the order of occurrence of the redundant indices. The pairs selected for embedding contain the index values to be used in the rearrangement. It is important to realize that the numerical values used for embedding data are the index values already occurring in the compression representation. The embedding process alters the entropy in the DCT or WSQ coefficients slightly, but the efficiency of compression is largely unaffected by embedding additional data into the indices values.

In the embedding process illustrated here, the coefficients calculated by the compression algorithm are manipulated in the order of the compression scheme used to generate the representation, JPEG and WSQ for the examples herein. The embedding process flows concurrently through the sequence of auxiliary data bits and the compression indices. Upon identifying a compression index matching one of the pair table values, the bit in the sequence of auxiliary data is examined to determine if the index is set to the **pair[k].i** value (embedding a 0), or set to the **pair[k].j** value (embedding a 1). The pair table is processed sequentially, in the order found after it was randomized by the code in Listing 11.

Listing 12 illustrates the code fragment that performs the actual embedding. The routine **embed_data_block()** embeds data into the block of data passed by the unsigned character pointer variable **block.** The loop index **j** increments by 2 each pass through the loop starting at line 20 in listing 12. The **block** pointer is used to extract the 16-bit integer into the variable **index** in line 21. Line 22 begins the loop index **k**, searching the values of the **pair[]** data structure. When a pair element is found to match the **index** value, embedding is permitted. The auxiliary data bit to be embedded is returned by the external routine **aux_bit().** For a one bit, the **index** variable is set to **pair[k].j** and for a zero bit, the **index** variable is set to **pair[k].i.** After setting the **index** variable, the proper two bytes in the **block** data array are loaded with the **index** variable value. The embedding proceeds as the index **j** strides through the compression representation indices.

Listing 12

```
/*********************************************************************************************
 *          embed_data_block:  Routine to embed data into a block of integers using
 *                                                the pairs in the structure pair.
 *          INPUT:    buffer, pointer to the block of integers
                               i,       integer no. of integers in the block
                               pair,    pointer to index pair structures
                               no_pairs, number of pair structures
            RETURN:nothing
            AUTHOR:            Copyright (c) University of California, 1995
                               M. T. Sandford II
 *********************************************************************************************/


void embed_data_block(unsigned char *buffer,int i,struct PAIRS *pair,int no_pairs) {
short index;
int j;
int k;
```

Listing 12, continued

```
float test;

  for(j=0;j<i;j+=2)    {
                      memcpy(&index,buffer+j,2);
                      for(k=0;k<no_pairs;k++) {
                              if(index==pair[k].i || index==pair[k].j) {
                                      embed_count++;
                                      if(aux_bit()) {        /* aux. bit is a 1 */
                                              index = pair[k].j;
                                              }
                                      else {                 /* aux. bit is a 0 */
                                              index = pair[k].i;
                                              }
                                      memcpy(buffer+j,&index,2);
                                      }
                              }
                      }
return;
}
```

7.5 Extracting compression-embedded data

The extraction of embedded data is accomplished by reversing the process used to embed the auxiliary bit stream. A histogram analysis of the compression representation containing embedded data will reveal the candidate pairs for extraction for only the case where the individual statistical frequencies of occurrence of the index values are unchanged by the embedding process. For most data, this is unlikely, and the pair value table recovered by an analysis is different from the one used for embedding.

Owing to the uncertainty introduced by randomizing the order of the pair table, the embedded data are more secure against unauthorized extraction from the compression representation. Indeed, detecting the presence of embedded data is difficult, if not impossible, because the only metric to use for such detection is a statistical analysis of the compression representation indices. Embedding affects the statistical properties only slightly and leaves no characteristic signature revealing the presence of embedded information.

With the pair table known, extraction consists of sequentially testing the index values to recreate the output bit-stream for the header information, and the auxiliary data. In the present invention, the pair table is inserted into the compressed image file header, or appended to the file end, where it is available for the extraction process. Typically, the pair table ranges from a few to perhaps hundreds of index values in size. The maximum pair table size permitted depends upon the compression representation. For JPEG compression, Fig. 2 shows the index values concentrate near the origin and 50 to 100 pair values are adequate. For WSQ compressions of digital fingerprints, the index values spread over a large spectrum and more pairs are required.

7.6 Improving lossy compression with data embedding

Embedding auxiliary data into the compression representation slightly changes the statistical frequency of occurrence of the index values. If the auxiliary bit sequence is pseudo-random, the frequencies of occurrence for the index pairs i and j are nearly equal after embedding. Intentionally equalizing the histogram reduces entropy somewhat, and the entropy coding portion of the compression algorithm is found to operate with slightly greater efficiency, increasing the effectiveness of the lossy compression method. Thus, even if no auxiliary information is to be embedded, equalizing the statistical properties of the histogram pairs improves the compression ratio for lossy methods.

Because the expansion of the compressed data returns an approximation to the original information, the efficiency of the lossy compression is unaffected by embedding the additional information. The embedded information can be extracted to a parallel channel by hardware or software added to the compression expansion algorithm. The embedding of information in this manner does not increase the bandwidth required for the transmission of the compressed data because the embedded data reside in the coefficients chosen originally to represent the input data.

8. CONCLUSIONS

Digital data containing a noise component are common in consumer products, military systems, and in scientific and technical applications. Because modifying data containing embedded information prevents extracting all the information correctly, the embedding process can serve as an invisible signature, or validation for digital data. A variant of data embedding permits placing a digital watermark in electronic images. Thus, digital images can be authenticated using the embedding method. For forensic applications, invisible embedded information can provide a chain of evidence that normally requires extensive paper trails.

Numerous electronic communication applications are possible. Ciphertext evident in a communication channel signifies protected information, but the same information can be embedded into routine digital data without noticeable effect. Thus, with data embedding, communications through public networks where encryption is not permitted resist analysis and monitoring. Previously open communications links have new potential for secure, covert communications.

Maps generated for military applications can be sent in digital form through open communications lines, but can easily contain embedded targeting information and overlays of battle order. The amount of information requiring protection is reduced in volume by data embedding, because only the unique key bytes must be protected. Indeed, maps can be released to, or even entrusted for transmission by commercial communication channels without compromise of their additional, embedded data.

Digital images can be watermarked using data embedding. A mask, or pattern, is followed to modify the content of selected pixels in the image. The modified pixels watermark the data, permitting commercial distribution without encryption or scrambling. Data embedding provides the means to restore the original image, because the pixels modified by watermarking are embedded into the noise component of the image. Using the embedding key and the mask, or pattern algorithm, the embedded information is extracted and the pixels in the watermark restored to their original content. Distribution of protected commercial imagery is reduced to key management without the need for additional media or image transmissions.

Data embedding can be used to convey information subject to Privacy Act controls. Patient identification and medical history can be embedded into a digital X-ray. Embedding medical information into digital X-rays, EKG, EEG, or MRI data provides a simple, easy way to bind privileged information with patient data and, simultaneously to provide authentication of the data. Commercial medical instruments can use data embedding, in combination with public-key encryption of the embedding key, to generate data that can be examined by anyone, yet extracted for privileged information by proper authorities. Data embedding is primarily a software application and it can therefore be implemented in many different types of existing hardware simply by modifying the programs used to process the data. Alternatively, the data embedding algorithm can be implemented in custom circuitry.

9. ACKNOWLEDGMENTS

We thank J. N. Stewart for his assistance in improving the coding of the two-color bitmap data embedding algorithms beyond the forms given in this paper. Dr. S. S. Hecker, Director of the Los Alamos National Laboratory, provided much needed support to pursue our investigations of data embedding algorithms. Mr. Milton Wyrick schooled us in the importance and methods of patenting software. Ms. Joyce Capell provided the opportunity to present our methods.

10. REFERENCES

1. M. T. Sandford II and T. G. Handel, "BMPEMBED: A Data Embedding Demonstration Application Program, Ver. 1.51," privately publ. software: Univ. of Calif. LANL, May 1994.

2. J. Levine, *Programming for Graphics Files* , publ. J. Wiley & Sons: New York, NY, 1994.

3. M. Luse, "The BMP Format," Dr. Dobb's J., **19**, 18, 1994.

4. B. Schneier, *Applied Cryptography Protocols, Algorithms, and Source Code in C* , publ. J. Wiley & Sons:New York, NY, 1994.

5. EXP Computer Corp., *User's Manual for EXP Fax/Data Modem* , 1.1 Edition, publ. EXP Computer, Inc.: Syosset, NY, 1993.

6. Microsoft Corp., *User's Guide, Microsoft Word*, Ver. 6.0, publ. Microsoft Press: Redmond, WA, 1994.

7. Bit Software, Inc., *BITFAX for Windows User's Guide*, 2nd ed., publ. Bit Software, Inc.: Fremont, CA, 1993.

8. Handmade Software, Inc., *Image Alchemy User's Manual*, Ver. 1.8, publ. Handmade Software, Inc.: Los Gatos, CA, 1995.

9. G. K. Wallace, "The JPEG Still Picture Compression Standard," Comm. of the ACM, **34** no. 4, pp. 30-44, 1991.

10. M. Nelson, *The Data Compression Book*, publ. M&T Books: Redwood City, CA, ISBN 1-55851-216-0, 1991.

11. W. B. Pennebaker and J. L. Mitchell, *JPEG Still Image Data Compression Standard*, publ.Van Nostrand Reinhold: NY, ISBN 0-442-01272-1, 1993.

12. J. N. Bradley and C. M. Brislawn, "The wavelet/scalar quantization standard for digital fingerprint images," *Proc. of the 1994 IEEE Intern. Symp. on Circuits and Systems,* **3**, pp. 205-208, 1994.

13. J. N. Bradley, C. M. Brislawn, and T. E. Hopper, "The FBI wavelet/scalar quantization standard for gray-scale fingerprint image compression," *Proc. SPIE*, **1961**, pp. 293-304, April, 1993.

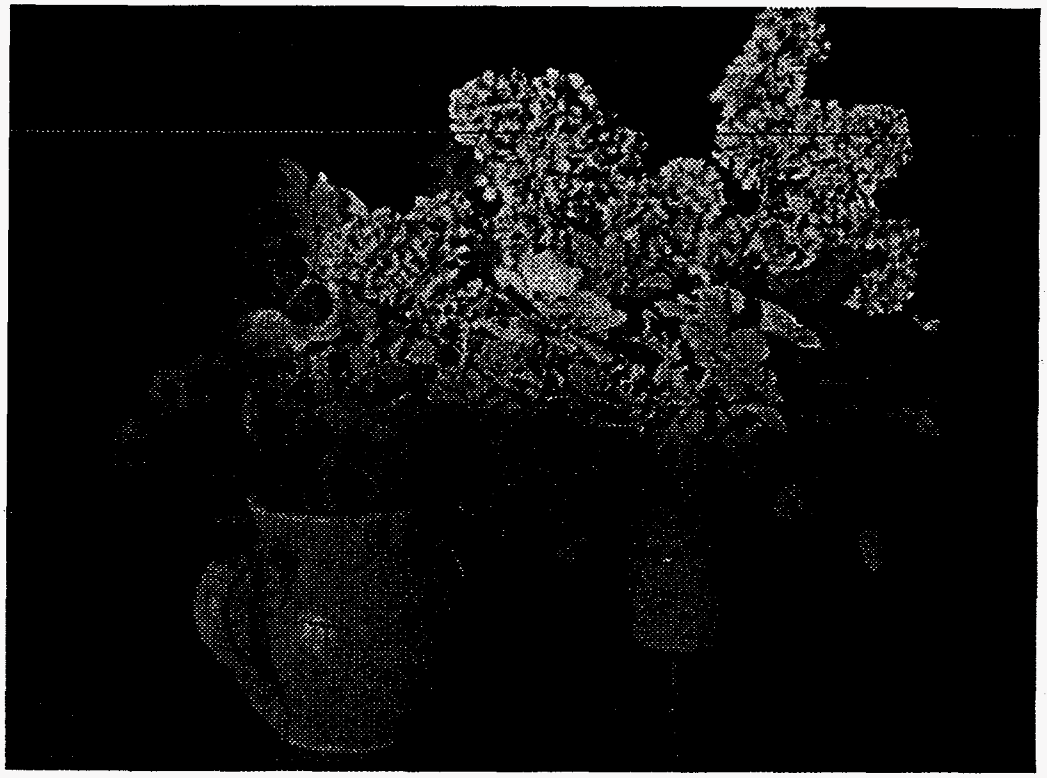


FIGURE 1 Greyscale photograph

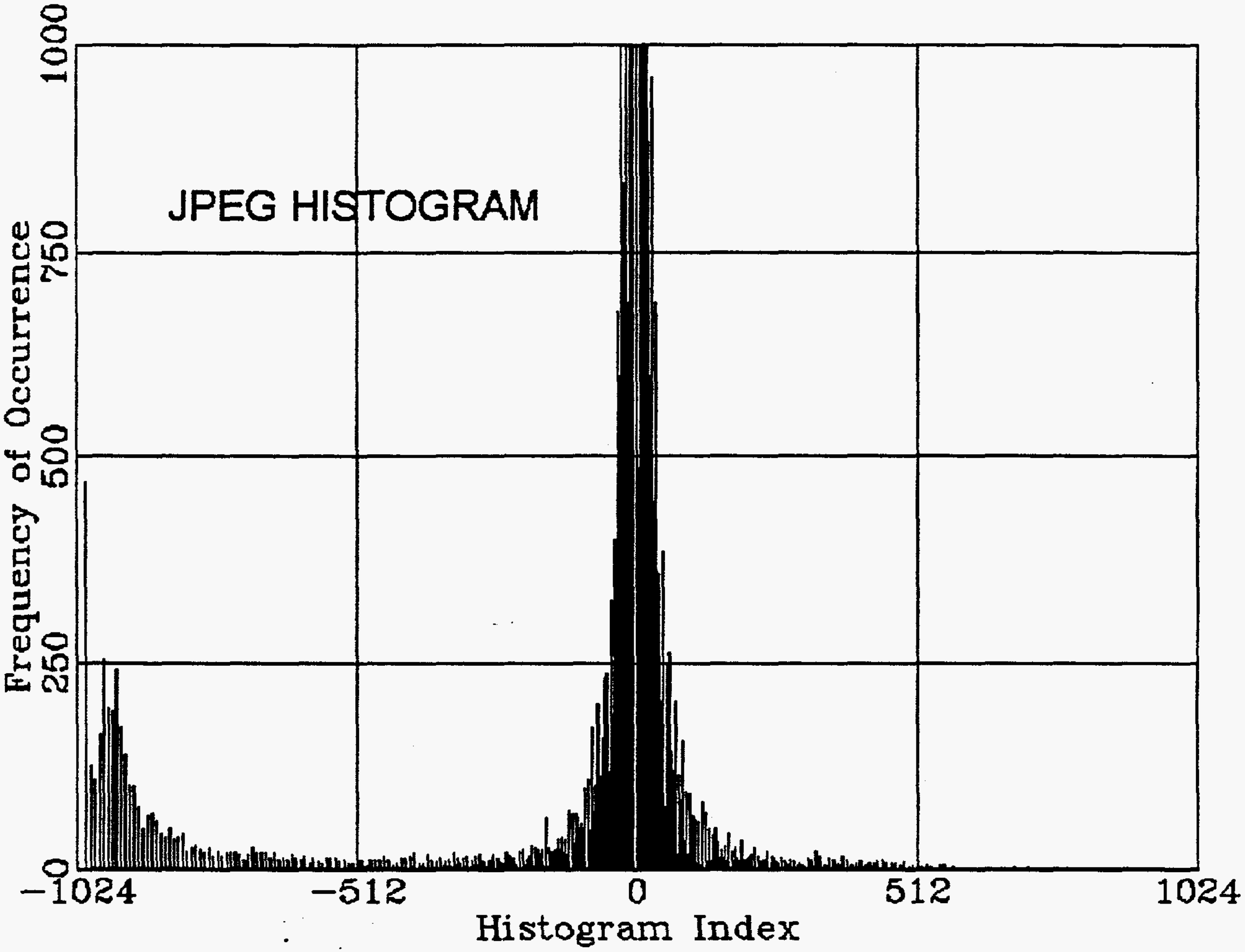


FIGURE 2 JPEG indices histogram of photograph

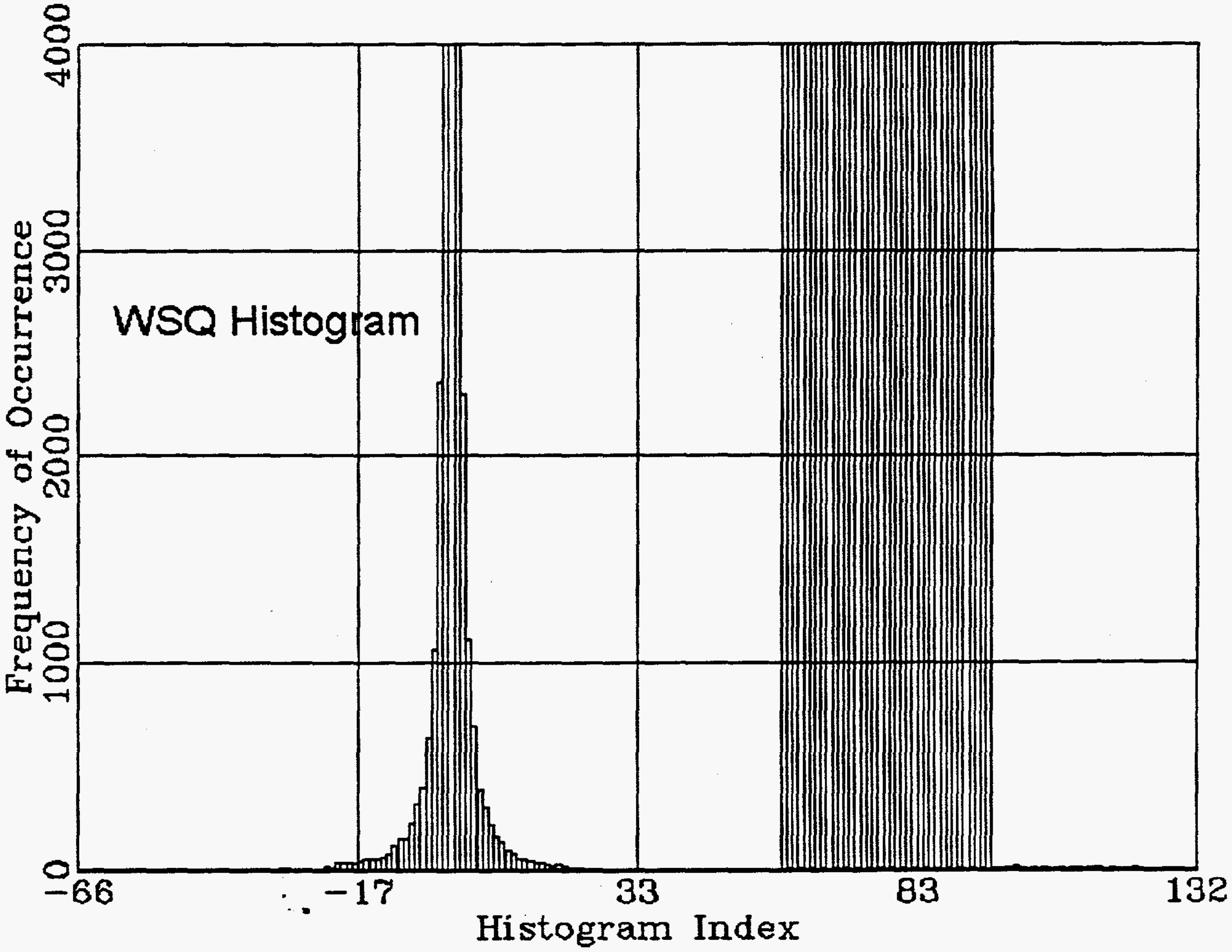


FIGURE 3 Wavelet indices histogram of photograph