Tiled Image Convention for Storing Compressed Images in FITS Binary Tables

Richard L. White, STScI
Perry Greenfield, STScI
William Pence, NASA/GSFC
Doug Tody, NRAO
Rob Seaman, NOAO

Version 2.3
2 July 2013

1 General Description

This document describes a convention for compressing n-dimensional images and storing the resulting byte stream in a variable-length column in a FITS binary table. The FITS file structure outlined here is independent of the specific data compression algorithm that is used. The implementation details for 4 widely used compression algorithms are described here, but any other compression technique could also be supported by this convention.

The general principle used in this convention is to first divide the n-dimensional image into a rectangular grid of subimages or 'tiles'. Each tile is then compressed as a block of data, and the resulting compressed byte stream is stored in a row of a variable length column in a FITS binary table. By dividing the image into tiles it is generally possible to extract and uncompress subsections of the image without having to uncompress the whole image. The default tiling pattern treats each row of a 2-dimensional image (or higher dimensional cube) as a tile, such that each tile contains `NAXIS1` pixels. This default many not be optimal for some applications or compression algorithms, so any other rectangular tiling pattern may be defined using the `ZTILEn` keywords that are described below. In the case of relatively small images, it may be sufficient to compress the entire image as a single tile, resulting in an output binary table with 1 row. In the case of 3-dimensional data cubes, it may be advantageous to treat each plane of the cube as a separate tile if application software typically needs to access the cube on a plane by plane basis.

2 Keywords

The following keywords are defined by this convention for use in the header of the FITS binary table extension to describe the structure of the compressed image.

- `ZIMAGE` (required keyword) This keyword must have the logical value T. It indicates that the FITS binary table extension contains a compressed image and that logically this extension should be interpreted as an image and not as a table.

- `ZCMPTYPE` (required keyword) The value field of this keyword shall contain a character string giving the name of the algorithm that must be used to decompress the image. Currently, values of `GZIP_1`, `GZIP_2`, `RICE_1`, `PLIO_1`, and `HCOMPRESS_1` are reserved, and the corresponding algorithms are described in a later section of this document. The value `RICE_ONE` is also reserved as an alias for `RICE_1`.

- `ZBITPIX` (required keyword) The value field of this keyword shall contain an integer that gives the value of the `BITPIX` keyword in the uncompressed FITS image.

- ZNAXIS (required keyword) The value field of this keyword shall contain an integer that gives the value of the NAXIS keyword in the uncompressed FITS image.

- ZNAXISn (required keywords) The value field of these keywords shall contain a positive integer that gives the value of the NAXISn keywords in the uncompressed FITS image.

- ZTILEn (optional keywords) The value of these indexed keywords (where n ranges from 1 to ZNAXIS) shall contain a positive integer representing the number of pixels along axis n of the compression tiles. Each tile of pixels is compressed separately and stored in a row of a variable-length vector column in the binary table. The size of each image dimension (given by ZNAXISn) is not required to be an integer multiple of ZTILEn, and if it is not, then the last tile along that dimension of the image will contain fewer image pixels than the other tiles. If the ZTILEn keywords are not present then the default 'row by row' tiling will be assumed such that ZTILE1 = ZNAXIS1, and the value of all the other ZTILEn keywords equals 1.

  The compressed image tiles are stored in the binary table in the same order that the first pixel in each tile appears in the FITS image; the tile containing the first pixel in the image appears in the first row of the table, and the tile containing the last pixel in the image appears in the last row of the binary table.

- ZNAMEn and ZVALn (optional keywords) These pairs of optional array keywords (where n is an integer index number starting with 1) supply the name and value, respectively, of any algorithm-specific parameters that are needed to compress or uncompress the image. The value of ZVALn may have any valid FITS datatype. The order of the compression parameters may be significant, and may be defined as part of the description of the specific decompression algorithm.

- ZMASKCMP (optional keyword) Used to record the name of the image compression algorithm that was used to compress the optional null pixel data mask. See the"Preserving undefined pixels with lossy compression" section for more details.

- The following 8 optional keywords are defined to store a verbatim copy of the the value and comment fields of the corresponding keywords in the original uncompressed FITS image. These keywords can be used to reconstruct an identical copy of the original FITS file when the image is uncompressed.

  - ZSIMPLE - preserves the original SIMPLE keyword
  - ZTENSION - preserves the original XTENSION keyword
  - ZEXTEND - preserves the original EXTEND keyword
  - ZBLOCKED - preserves the original BLOCKED keyword
  - ZPCOUNT - preserves the original PCOUNT keyword
  - ZGCOUNT - preserves the original GCOUNT keyword
  - ZHECKSUM - preserves the original CHECKSUM keyword
  - ZDATASUM - preserves the original DATASUM keyword

  The ZSIMPLE, ZEXTEND, and ZBLOCKED keywords may only be used if the original uncompressed image was contained in the primary array of the FITS file. The ZTENSION, ZPCOUNT, and ZGCOUNT keywords may only be used if the original uncompressed image was contained in in IMAGE extension.

- ZQUANTIZ (optional keyword) This keyword records the name of the algorithm that was used to quantize floating-point image pixels into integer values which are then passed to the compression algorithm, as discussed further in section 4 of this document.

- ZDITHER0 (optional keyword) The value field of this keyword shall contain an integer that gives the seed value for the random dithering pattern that was used when quantizing the floating-point pixel values. The value may range from 1 to 10000, inclusive. See section 4 for further discussion of this keyword.

- Other Keywords The FITS header of the compressed image may contain other optional keywords. If a FITS primary array or IMAGE extension is compressed using the convention described here, it is recommended that all the keywords in the header of the original image, except for the mandatory keywords mentioned above, be copied verbatim and in the same order into the header of the binary table extension that contains the compressed image. All these keywords will have the same meaning and interpretation as they did in the original image, even in cases where the keyword is not normally expected to occur in the header of a binary table extension (e.g., the BSCALE and BZERO keywords, or the World Coordinate System keywords such as CTYPEn, CRPIXn and CRVALn).

3 Columns

The following columns in the FITS binary table are defined by this convention. The order of the columns in the table is not significant. The column names (given by the TTYPEn keyword) are shown here in upper case letters, but the case is not significant.

Note regarding the variable-length columns: The COMPRESSED_DATA, GZIP_COMPRESSED_DATA, and UNCOMPRESSED_DATA columns described below will normally use the '1P' variable-length array FITS column format if the size of the heap in the compressed FITS file is less than about 2.1 GB in size. If the the heap is larger than 2.1 GB, then the '1Q' format (which uses 64-bit pointers) must be used.

- COMPRESSED_DATA (required column)

  Each row of this variable-length column contains the byte stream that is generated as a result of compressing the corresponding image tile. The datatype of the column (as given by the TFORMn keyword) will generally be either '1PB', '1PI', or '1PJ' (or the equivalent '1Q' format), depending on whether the compression algorithm generates an output stream of 8-bit bytes, 16-bit integers, or 32-bit integers, respectively.

- GZIP_COMPRESSED_DATA (optional column)

  When using the quantization method to compress floating-point images that is described in Section 4, it sometimes may not be possible to quantize some of the tiles (e.g., if the range of pixels values is too large or if most of the pixels have the same value and hence the calculated RMS noise level in the tile is close to zero). There also may be other rare cases where the nominal compression algorithm can not be applied to certain tiles. In these cases, one may use an alternate technique in which the raw pixel values are losslessly compressed with the GZIP algorithm and the resulting byte stream is stored in the GZIP_COMPRESSED_DATA column (with a '1PB' or '1QB' variable-length array column format). The corresponding COMPRESSED_DATA column for these tiles must contain a null pointer.

- `UNCOMPRESSED_DATA` (optional column)

  Use of this column is no longer recommended, but it may exist in older compressed image files that were created before support for the `GZIP_COMPRESSED_DATA` column (describe above) was added to this convention in May 2011. This variable length column contains the uncompressed pixels for any tiles that cannot be compressed with the normal method. The datatype of this column should correspond to the datatype of the original image as shown in the following table:

Datatype	`BITPIX`	`TFORMn`
byte	8	'1PB' or '1QB'
short int	16	'1PI' or '1QI'
long int	32	'1PJ' or '1QJ'
float	-32	'1PE' or '1QE'
double	-64	'1PD' or '1QD'

  A compressed image may contain either the `UNCOMPRESSED_DATA` column or the `GZIP_COMPRESSED_DATA` column, but not both.

- `ZSCALE` and `ZZERO` (optional floating-point columns)

  When using the quantization method to compress floating-point images that is described in Section 4, these 2 columns store the linear scale factor and the zero point offset, respectively, that are used to scale the floating-point pixel values into integers via,

  $$I_i = ROUND((F_i - ZZERO)/ZSCALE) \tag{1}$$

  where I_i and F_i are the integer and floating-point values, respectively and the ROUND function rounds the result to the nearest integer value. The array of integer tile pixel values is then compressed using the algorithm that is specified by the `ZCMPTYPE` keyword and the resulting compressed byte stream is stored in the `COMPRESSED_DATA` column.

  The `ZSCALE` and `ZZERO` columns should not be confused with the reserved `BSCALE` and `BZERO` keywords which may be present in integer FITS images (which have BITPIX = 8, 16, or 32). Any such integer images should normally be compressed without any further scaling, and the `BSCALE` and `BZERO` keywords should be copied verbatim into the header of the binary table containing the compressed image.

- `ZBLANK` (optional column or keyword)

  When using the quantization method to compress floating-point images that is described in Section 4, this column is used to store the integer value that represents undefined pixels (if any) in the scaled integer pixel values. These pixels have an IEEE NaN value (Not a Number) in the uncompressed floating-point image. The recommended value for `ZBLANK` is -2147483648 (the largest negative 32-bit integer).

  If the same null value is used in every tile of the image, then `ZBLANK` may be given as a header keyword instead of a table column. If there are no undefined pixels in the image then `ZBLANK` is not required. If the uncompressed image has an integer datatype (`ZBITPIX` > 0) then the reserved `BLANK` keyword, which already serves this purpose, should be used instead of `ZBLANK`.

- `NULL_PIXEL_MASK` (optional column)

  When using some image compression techniques that do not exactly preserve integer pixel values, it it may be necessary to store a compressed image mask along with the compressed image itself, to record the location of the undefined pixels in the image. This `NULL_PIXEL_MASK` column may be used for this purpose. See the "Preserving undefined pixels with lossy compression" section for more details.

- `Other Columns` Any number of other columns may be present in the table to supply other parameters that relate to each image tile.

4 Quantization of Floating-Point Data

While floating-point format images may be losslessly compressed (using gzip, since Rice and H-compress only compress integer arrays), these images often do not compress very well because the pixel values are too noisy; the less significant bits in the mantissa of the pixel values effectively contain incompressible random bit patterns. In order to achieve higher compression, one needs to remove some of this noise, but without losing the useful information content. One commonly used technique for reducing the noise is to scale the floating-point values into quantized integers using Eq. 1, and using the `ZSCALE` and `ZZERO` columns to record the 2 scaling coefficients that are used for each tile. Note that the absence of these 2 columns in a tile-compressed floating-point image is an indication that the image was not scaled and was instead losslessly compressed.

The main challenge in quantizing the image in this way is in choosing an appropriate scaling factor. If it is too large, one undersamples the pixel values resulting in a loss of information in the image. If it is too small, however, it preserves too much of the noise (or even amplifies the noise) in the pixel values, resulting in poor compression.

An effective scaling algorithm for preserving a specified amount of noise in each pixel value is described by White and Greenfield (in the Proceedings of the 1998 ADASS VIII conference) and by Pence, Seaman, and White, PASP 121, 414 (2009). With this method, the ZSCALE value (which is numerically equal to the spacing between adjacent quantization levels) is calculated to be some fraction, Q, of the RMS noise as measured in background regions of the image. It can be shown that the number of binary bits of noise that are preserved in each pixel value is given by $log_2(Q) + 1.792$. For example, using Q = 8 (so that the quantized levels have a spacing of 1/8th of the background RMS noise value) produces a quantized image that preserves about 4.8 bits of noise in each pixel. Specifying the quantization level relative to the amount of noise in the image in this way produces comparable quality images regardless of the noise level. Q is directly related to the compressed file size: decreasing Q by a factor of 2 will decrease the file size by about 1 bit/pixel. In order to achieve the greatest amount of compression, one should use the smallest value of Q that still preserves the required amount of photometric and astrometric precision in the image.

One potential problem when applying this scaling method to astronomical images, in particular, is that it can lead to a systematic bias in the measured intensities in faint parts of the image, such as in the background sky. As the image is quantized more coarsely, the measured intensity of the background regions of the sky will tend to be biased towards the nearest quantize level. One very effective technique for minimizing this potential bias is to "dither" the quantized pixel values by introducing random noise during the quantization process. So instead of simply scaling every pixel value in the same way using Eq. 1, one randomizes the quantized levels by using this slightly modified equation:

$$I_i = ROUND(((F_i - ZZERO)/ZSCALE) + R_i - 0.5) \tag{2}$$

where R_i is a random number between 0.0 and 1.0, and the 0.5 term is subtracted so that the mean quantity is equal to 0. Then restoring the floating-point value, the same random number is used with the inverse formula

$$F_i = ((I_i - R_i + 0.5) * ZSCALE) + ZZERO \tag{3}$$

This technique, which is called 'subtractive dithering' in the signal processing literature (e.g., "Quantization Noise" by Widrow and Kollar), has the effect of dithering the zero-point of the quantization grid on a pixel by pixel basis without adding any actual noise to the image. The net effect of this is that the mean (and median) pixel value in faint regions of the image more closely approximate the value in the original unquantized image than if all the pixels are scaled without dithering. This can significantly increase the precision when measuring the net flux from faint sources in the compressed image.

The key requirement when using this subtractive dithering technique is that the exact same random number sequence must be used when quantizing the pixel values to integers, and when restoring them to floating point values. While most computer languages supply a function for generating random numbers, these functions are not guaranteed to generate the same sequence of numbers every time. Accordingly, we define a specific algorithm here for generating a repeatable sequence of pseudo random numbers in Appendix A.

4.1 Dithering Algorithms

When quantizing floating point images, one may choose from the 3 currently defined dithering algorithms as specified by the value of the `ZQUANTIZ` keyword, as described in the following sections.

4.1.1 `ZQUANTIZ= 'NO_DITHER'`

This is the simplest option in which no dithering is performed. The floating-point pixels are simply quantized using Eq. 1. This option should be assumed if the `ZQUANTIZ` keyword is not present in the header of the compressed floating-point image.

4.1.2 `ZQUANTIZ= 'SUBTRACTIVE_DITHER_1'`

The steps in this dithering option are as follows:

1. Generate a sequence of 10000 single-precision floating-point random numbers, RN, with a value between 0.0 and 1.0, using the algorithm given in Appendix A. Since it could be computationally expensive to generate a unique random number for every pixel of large images, we repeatedly recycle through this 'look up table' of random numbers.

2. Choose an integer in the range 1 to 10000 to serve as an initial seed value for creating a unique sequence of random numbers from the array that was calculated in the previous step. The purpose of this is to reduce the chances of applying the same dithering pattern to 2 images that are subsequently subtracted from each other (or co-added), because the benefits of randomized dithering are lost if all the pixels are dithered in phase with each other. The

exact method for computing this seed integer is not important as long as the value is chosen more or less randomly. For example, one might calculate the seed value based on the system clock time when the image is compressed, or based on the checksum of all the pixel values in the first image tile that is compressed. However, beware of using the checksum method for choosing the seed value in cases where the first row/tile of all the images in a dataset are identical, as can happen if all the images have a border of zero or null valued pixels around the actual image.

3. Write the integer seed value that was selected in the previous step as the value of the `ZDITHER0` keyword in the header of the compressed image. This value is required to recompute the same dithering pattern when uncompressing the image.

4. Before quantizing each tile of the floating point image, calculate an initial value for 2 offset parameters, I0 and I1, with the following formulae:

$$I0 = modulo(TILE\_NUMBER - 1 + ZDITHER0, 10000) \tag{4}$$
$$I1 = INT(RN(I0) * 500.) \tag{5}$$

   where TILE_NUMBER is the row number in the binary table that is used to store the compressed bytes for that tile, `ZDITHER0` is that value of that keyword, and RN(I0) is the value of the (I0)th random number in the sequence that was computed in the first step. Note that I0 has a value in the range 0 to 9999 and I1 has a value in the range 0 to 499. This method for computing I0 and I1 was chosen so that a different sequence of random number is used to compress successive tiles in the image, and so that the sequence of I1 values has a length of order 100 million elements before repeating.

5. Now quantize each floating-point pixel in the tile using Eq. 2 and using random number RN(I1) for the first pixel. Increment the value of I1 for each subsequent pixel in the tile. If the value of I1 reaches the upper limit of 10000, then increment the value of I0 and recompute I1 from Eq. 5. If the value of I0 also reaches the upper limit of 10000, then reset I0 to 0.

   If the floating-point pixel has an IEEE NaN value, then it is not quantized or dithered and instead it is set to the reserved integer value that is specified by the `ZBLANK` keyword. For consistency, the value of I1 should also be incremented in this case even though it is not used.

6. Compress the array of quantized integers using the lossless algorithm that is specified by the `ZCMPTYPE` keyword (use Rice by default).

7. Write the compressed array of bytes into the `COMPRESSED_DATA` column in the appropriate row of the binary table corresponding to that tile.

8. Write the linear scaling and zero point values that were used in Eq. 2 for that tile into the `ZSCALE`) and `ZZERO` columns in the same row of the binary table.

9. Repeat Steps 4 through 8 for each tile of the image.

While the above dithering algorithm is clearly not unique, we present it here as a well defined method that should be possible to implement in almost any computer language. It should be noted that an image that is quantized using this technique can still be unquantized using the simple linear scaling function given by Eq. 1. The only side effect in this case is to introduce slightly more noise in the image than if the full subtractive dithering algorithm were applied.

4.1.3 ZQUANTIZ= 'SUBTRACTIVE_DITHER_2'

This dithering algorithm is identical to the SUBTRACTIVE_DITHER_1 algorithm described above, except that any pixels in the floating-point image that are equal to 0.0 are represented by the reserved value -2147483647 in the quantized integer array. When the image is subsequently uncompressed and unscaled, these pixels are restored to their original value of 0.0. This dithering option is useful if the zero-valued pixels have special significance to the data analysis software, so that the value of these pixels must not be dithered.

5 Preserving undefined pixels with lossy compression

The null pixels in integer images are flagged by a reserved BLANK value and will be preserved if a lossless compression algorithm is used. If the image is compressed with a lossy algorithm, however (e.g., H-Compress with a scale factor greater than 1), then some other technique must be used to identify the null pixels in the image.

The recommended method of recording the null pixels when a lossy compression algorithm is used is to create an integer data mask with the same dimensions as the image tile. Set the null pixels to 1 and all the other pixels to 0, then compress the mask array using a lossless algorithm such as PLIO or GZIP. Store the compressed byte stream in a variable-length array column called 'NULL_PIXEL_MASK' in the row corresponding to that image tile. The ZMASKCMP keyword should be used to record the name of the algorithm used to compress the data mask (e.g., RICE_1). The data mask array pixels will be assumed to have the shortest integer datatype that is supported by the compression algorithm (i.e., usually 8-bit bytes).

When uncompressing the image tile, the software must check if the corresponding compressed data mask exists with a length greater than 0, and if so, then uncompress the mask and set the corresponding undefined pixels in the image array to the appropriate value (as given by the BLANK keyword).

6 Currently Implemented Compression Algorithms

This section describes the 4 compression algorithms that are currently supported in the CFITSIO implementation of this tiled image compression convention (available from the HEASARC web site). This does not imply that other implementations of this convention must support these same algorithms, nor does it limit other implementations from supporting other compression algorithms.

6.1 Rice compression algorithm

If ZCMPTYPE = 'RICE_1' then the Rice algorithm is used to compress and uncompress the image pixels. The Rice algorithm (Rice, R. F., Yeh, P.-S., and Miller, W. H. 1993, in Proc. of the 9th AIAA Computing in Aerospace Conf., AIAA-93-4541-CP, American Institute of Aeronautics and Astronautics) is simple and very fast, compressing or decompressing 10^7 pixels/sec on modern workstations. It requires only enough memory to hold a single block of 16 or 32 pixels at a time. It codes the pixels in small blocks and so is able to adapt very quickly to changes in the input image statistics (e.g., Rice has no problem handling cosmic rays, bright stars, saturated pixels, etc.).

The block size that is used should be recorded in the compressed image header with

```
ZNAMEn  = 'BLOCKSIZE'
ZVALn   = 16 or 32
```

If these keywords are absent, then a default blocksize of 32 should be assumed.

The number of 8-bit bytes in each original integer pixel value should be recorded in the compressed image header with

```
ZNAMEn  = 'BYTEPIX'
ZVALn   = 1, 2, 4, or 8
```

If these keywords are absent, then the default value of 4 bytes per pixel (32 bits) should be assumed..

6.2 GZIP compression algorithm

If ZCMPTYPE = 'GZIP_1' then the gzip algorithm is used to compress and uncompress the image pixels. Gzip is the compression algorithm used in the free GNU software utility of the same name. It was created by Jean-loup Gailly and Mark Adler and is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. DEFLATE was intended as a replacement for LZW and other patent-encumbered data compression algorithms which, at the time, limited the usability of compress and other popular archivers. Further information about this compression technique is readily available on the Internet. The gzip algorithm has no associated parameters that need to be specified with the ZNAMEn and ZVALn keywords.

If ZCMPTYPE = 'GZIP_2' then the bytes in the array of image pixel values are shuffled into decreasing order of significance before being compressed with the gzip algorithm. In other words, bytes are shuffled so that the most significant byte of every pixel occurs first, in order, followed by the next most significant byte, and so on for every byte. Since the most significan bytes of the pixel values often have very similar values, grouping them together in this way often achieves better net compression of the array. This is usually especially effective when compressing floating-point arrays.

6.3 IRAF PLIO compression algorithm

If ZCMPTYPE = 'PLIO_1' then the IRAF PLIO (Pixel List) algorithm is used to compress and uncompress the image pixels. The PLIO algorithm was developed to store integer-valued image masks in a compressed form. Typical uses of image masks are to segment images into regions, or to mark bad pixels. Such masks often have large regions of constant value hence are highly compressible. The compression algorithm used is based on run-length encoding, with the ability to dynamically follow level changes in the image, allowing a 16-bit encoding to be used regardless of the image depth. The worst case performance occurs when successive pixels have different values. Even in this case the encoding will only require one word (16 bits) per mask pixel, provided either the delta intensity change between pixels is usually less than 12 bits, or the mask represents a zero floored step function of constant height. The worst case cannot exceed npix*2 words provided the mask depth is 24 bits or less.

A good compromise between storage efficiency and efficiency of runtime access, while keeping things simple, is achieved if we maintain the compressed line lists as variable length arrays of type short integer (16 bits per list element), regardless of the mask depth. A line list consists of a series of simple instructions which are executed in sequence to reconstruct a line of the mask. Each 16 bit instruction consists of the sign bit (not used at present), a three bit opcode, and twelve bits of data, i.e.:

```
+--+-----------+--------------------------+
|16|15       13|12                       1|
```

```
+--+-----------+-----------------------------+
|  |   opcode  |           data              |
+--+-----------------------------------------+
```

The significance of the data depends upon the instruction. The instructions currently implemented are summarized in the table below.

Instruction	Opcode	Description
ZN	00	Output N zeros
HN	04	Output N high values
PN	05	Output N-1 zeros plus one high value
SH	01	Set high value, absolute
IH,DH	02,03	Increment or decrement high value
IS,DS	06,07	Like IH-DH, plus output one high value

In order to reconstruct a mask line, the application executing these instructions is required to keep track of two values, the current high value and the current position in the output line. The detailed operation of each instruction is as follows:

ZN Zero the next N (=data) output pixels.

HN Set the next N output pixels to the current high value.

PN Zero the next N-1 output pixels, and set pixel N to the current high value.

SH Set the high value (absolute rather than incremental), taking the high 15 bits from the next word in the instruction stream, and the low 12 bits from the current data value.

IH,DH Increment (IH) or decrement (DH) the current high value by the data value. The current position is not affected.

IS,DS Increment (IS) or decrement (DS) the current high value by the data value, and step, i.e., output one high value.

The high value is assumed to be set to 1 at the beginning of a line, hence the IH,DH and IS,DS instructions are not normally needed for Boolean masks. If the length of a line segment of constant value or the difference between two successive high values exceeds 4096 (12 bits), then multiple instructions are required to describe the segment or intensity change.

6.4 H-Compress algorithm

Hcompress is an the image compression package written by Richard L. White for use at the Space Telescope Science Institute. Hcompress was used to compress the STScI Digitized Sky Survey and has also been used to compress the preview images in the Hubble Data Archive. Briefly, the method used is:

1. a wavelet transform called the H-transform (a Haar transform generalized to two dimensions), followed by

2. quantization that discards noise in the image while retaining the signal on all scales, followed by

3. quadtree coding of the quantized coefficients.

The technique gives very good compression for astronomical images and is relatively fast. The calculations are carried out using integer arithmetic and are entirely reversible. Consequently, the program can be used for either lossy or lossless compression, with no special approach needed for the lossless case (e.g. there is no need for a file of residuals.)

There are 2 user-defined parameters associated with the H-Compress algorithm: an integer scale factor that determines the amount of compression, and a Boolean parameter the specifies whether the image should be smoothed during the decompression operation, to reduce residual artifacts in the image.

- **Scale Factor.** The integer scale parameter determines the amount of compression. Scale = 0 or 1 leads to lossless compression, i.e. the decompressed image has exactly the same pixel values as the original image. If the scale factor is greater than 1 then the compression is lossy: the decompressed image will not be exactly the same as the original. For astronomical images, lossless compression is generally rather ineffective because the images have a good deal of noise, which is inherently incompressible. However, if some of this noise is discarded then the images compress very well. The scale factor determines how much of the noise is discarded. Setting scale to 2 times sigma, the RMS noise in the image, usually results in compression by about a factor of 10 (i.e. the compressed image requires about 1.5 bits/pixel), while producing a decompressed image that is nearly indistinguishable from the original. In fact, the RMS difference between the decompressed image and the original image will be only about 1/2 sigma. Experiments indicate that this level of loss has no noticeable effect on either the visual appearance of the image or on quantitative analysis of the image (e.g. measurements of positions and brightnesses of stars are not adversely affected.)

  Using a larger value for scale results in higher compression at the cost of larger differences between the compressed and original images. A rough rule of thumb is that if scale equals N sigma, then the image will compress to about 3/N bits/pixel, and the RMS difference between the original and the compressed image will be about N/4 sigma. This crude relationship is inaccurate both for very high compression ratios and for lossless compression, but it does at least give an indication of what to expect of the compressed images.

  For images in which the noise varies from pixel to pixel (e.g. CCD images, where the noise is larger for brighter pixels), the appropriate value for scale is determined by the RMS noise level in the sky regions of the image. For images that are essentially noiseless, any lossy compression is noticeable under sufficiently close inspection of the image, but some loss is nonetheless acceptable for typical applications. Note that the quantization scheme used in Hcompress is not designed to give images that appear as much like the original as possible to the human eye, but rather is designed to produce images that are as similar as possible to the original under quantitative analysis. Thus, the emphasis is on discarding noise without affecting the signal rather than on discarding components of the image that are not very noticeable to the eye (as may be done, for example, by JPEG compression.) The resulting compression scheme is not ideal for typical terrestrial images (though it is still a reasonably good method for those images), but is believed to be close to optimal for astronomical images.

  It is not necessary to know what scale factor was used when compressing the image in order to uncompress it, but it is still useful to record the value that was used. It is recommended that the `ZNAMEn` and `ZVALn)` pair of keywords be used for this purpose, with

  ```
  ZNAMEn = 'SCALE'
  ```

```
ZVALn  =  I
```

where I is the integer scale value.

- **Smoothing Flag.** At high compressions factors the decompressed image begins to appear blocky because of the way information is discarded. This blockiness ness is greatly reduced, producing more pleasing images, if the image is smoothed slightly during decompression. When done properly, the smoothing will not affect any quantitative photometric or astrometric measurements derived from the compressed image. Of course, the smoothing should never be applied when the image has been losslessly compressed with a scale factor (defined above) of 0 or 1.

  The smoothing option only needs to be specified when uncompressing the image, however, in many cases, this can best be determined by the person or project that creates the compressed image files. Thus it is recommended that the smoothing flag be specified in the compressed image header with the `ZNAMEn` and `ZVALn` keywords with

  ```
  ZNAMEn = 'SMOOTH'
  ZVALn  =  0 or 1
  ```

  A value of 0 means no smoothing, and any other value means smoothing is recommended. This should be regarded as only a recommendation which the image decompression program may override.

A paper describing Hcompress was published in the Proceedings of the NASA Space and Earth Science Data Compression Workshop, ed. James C. Tilton, Snowbird, Utah, March 1992. This paper is reproduced in the Appendix B of this document.

A Random Number Generator

This portable random number generator algorithm comes from the publication "Random number generators: good ones are hard to find", Communications of the ACM, Volume 31 , Issue 10 (October 1988) Pages: 1192 - 1201 which is available on the Web. This algorithm basically just repeatedly evaluates the function seed = (a * seed) mod m, where the values of a and m are shown below, but it is implemented in a way to avoid integer overflow problems.

```
int random_generator(void) {

/* initialize an array of random numbers */

  int ii;
  double a = 16807.0;
  double m = 2147483647.0;
  double temp, seed;
  float rand_value[10000];

  /* initialize the random numbers */
  seed = 1;
```

```
  for (ii = 0; ii < N_RANDOM; ii++) {
      temp = a * seed;
      seed = temp -m * ((int) (temp / m) );
      rand_value[ii] = seed / m;  /* divide by m to get value between 0 and 1 */
  }
}
```

If implemented correctly, the 10000th value of seed must equal 1043618065.

B High-Performance Compression of Astronomical Images

Richard L. White
Joint Institute for Laboratory Astrophysics, University of Colorado
Campus Box 440, Boulder, CO 80309
and
Space Telescope Science Institute
3700 San Martin Drive, Baltimore, MD 21218
rlw@stsci.edu

Summary

Astronomical images have some rather unusual characteristics that make many existing image compression techniques either ineffective or inapplicable. A typical image consists of a nearly flat background sprinkled with point sources and occasional extended sources. The images are often noisy, so that lossless compression does not work very well; furthermore, the images are usually subjected to stringent quantitative analysis, so any lossy compression method must be proven not to discard useful information, but must instead discard only the noise. Finally, the images can be extremely large. For example, the Space Telescope Science Institute has digitized photographic plates covering the entire sky, generating 1500 images each having 14000×14000 16-bit pixels. Several astronomical groups are now constructing cameras with mosaics of large CCDs (each 2048×2048 or larger); these instruments will be used in projects that generate data at a rate exceeding 100 MBytes every 5 minutes for many years.

An effective technique for image compression may be based on the H-transform/ (Fritze et al. 1977). The method that we have developed can be used for either lossless or lossy compression. The digitized sky survey images can be compressed by at least a factor of 10 with no noticeable losses in the astrometric and photometric properties of the compressed images. The method has been designed to be computationally efficient: compression or decompression of a 512×512 image requires only 4 seconds on a Sun SPARCstation 1. The algorithm uses only integer arithmetic, so it is completely reversible in its lossless mode, and it could easily be implemented in hardware for space applications.

1. Introduction

Astronomical images consist largely of empty sky. Compression of such images can reduce the volume of data that it is necessary to store (an important consideration for large scale digital sky surveys) and can shorten the time required to transmit images (useful for remote observing or remote access to data archives.)

Data compression methods can be classified as either "lossless" (meaning that the original data can be reconstructed exactly from the compressed data) or "lossy" (meaning that the uncompressed image is not exactly the same as the original.) Astronomers often insist that they can accept only lossless compression, in part because of conservatism, and in part because the familiar lossy compression methods sacrifice some information that is needed for accurate analysis of image data. However, since all astronomical images contain noise, which is inherently incompressible, lossy compression methods produce much better compression results.

A simple example may make this clear. One of the simplest data compression techniques is run-length coding, in which runs of consecutive pixels having the same value are compressed by storing the pixel value and the repetition factor. This method is used in the standard compression scheme for facsimile transmissions. Unfortunately, it is quite ineffective for lossless compression of astronomical images because even though the sky is *nearly* constant, the noise in the sky ensures that only very short runs of equal pixels occur. The obvious way to make run-length coding more effective is to force the sky to be exactly constant by setting all pixels below a threshold (chosen to be just above the sky) to the mean sky value. However, then one has lost any information about objects close to the detection limit. One has also lost information about local variations in the sky brightness, which severely limits the accuracy of photometry and astrometry on faint objects. Worse, there may be extended, low surface brightness objects that are not detectable in a single pixel but that are easily detected when the image is smoothed over a number of pixels; such faint structures are irretrievably lost when the image is thresholded to improve compression.

2. The H-transform

Fritze et al. (1977; see also Richter 1978 and Capaccioli et al. 1988) have developed a much better compression method for astronomical images based on what they call the *H-transform* of the image. A similar transform called the S-transform has also been used for image compression (Blume & Fand 1989). The H-transform is a two-dimensional generalization of the Haar transform (Haar 1910). The H-transform/ is calculated for an image of size $2^N \times 2^N$ as follows:

- Divide the image up into blocks of 2×2 pixels. Call the 4 pixels in a block a_{00}, a_{10}, a_{01}, and a_{11}.
- For each block compute 4 coefficients:

  $$h_0 = (a_{11} + a_{10} + a_{01} + a_{00})/2$$
  $$h_x = (a_{11} + a_{10} - a_{01} - a_{00})/2$$
  $$h_y = (a_{11} - a_{10} + a_{01} - a_{00})/2$$
  $$h_c = (a_{11} - a_{10} - a_{01} + a_{00})/2$$

- Construct a $2^{N-1} \times 2^{N-1}$ image from the h_0 values for each 2×2 block. Divide that image up into 2×2 blocks and repeat the above calculation. Repeat this process N times, reducing the image in size by a factor of 2 at each step, until only one h_0 value remains.

This calculation can be easily inverted to recover the original image from its transform. The transform is exactly reversible using integer arithmetic if one does not divide by 2 for the first set of coefficients. It is straightforward to extend the definition of the transform so that it can be computed for non-square images that do not have sides that are powers of 2. The H-transform can be performed in place in memory and is very fast to compute, requiring about $16M^2/3$ (integer) additions for a $M \times M$ image.

The H-transform is a simple 2-dimensional wavelet transform. It has several advantages over some other wavelet transforms that have been applied to image compression (e.g., Daubechies 1988). First, the transform can be performed entirely with integer arithmetic, making it exactly reversible. Consequently it can be used for either lossless or lossy compression (as indicated below) and one does not need a special technique for the case of lossless compression (as was required, e.g., for the JPEG compression standard.)

A second major advantage is that the H-transform is a native 2-dimensional wavelet transform. The standard 1-dimensional wavelet transforms are extended to two dimensions by transforming the image first along the rows, then along the columns. Unfortunately, this generates many wavelet coefficients that are high frequency (hence localized) in the x-direction but low frequency (hence global) in the y-direction. Such coefficients are counter to the philosophy of the wavelet transform: high-frequency basis functions should be confined to a relatively small area of the image.

Discarding these mixed-scale terms, which may be negligible compared to the noise, generates very objectionable artifacts around point sources and edges in the image. The H-transform, on the other hand, is a fully 2-dimensional wavelet transform, with all high frequency terms being completely localized. It is consequently more suitable for image compression and produces fewer artifacts.

A possible disadvantage of the H-transform is that other wavelet transforms take better advantage of the continuity of pixel values within images, so that they can produce higher compressions for very smooth images. However, for astronomical images (which are mostly flat sky sprinkled with point sources) the smoothness built into higher-order transforms can actually reduce the effectiveness of compression, because one must keep more coefficients to describe each point source.

3. Compression Using the H-transform

If the image is nearly noiseless, the H-transform is somewhat easier to compress than the original image because the differences of adjacent pixels (as computed in the H-transform) tend to be smaller than the original pixel values for smooth images. Consequently fewer bits are required to store the values of the H-transform coefficients than are required for the original image. For very smooth images the pixel values may be constant over large regions, leading to transform coefficients that are zero over large areas.

Noisy images still do not compress well when transformed, though. Suppose there is noise σ in each pixel of the original image. Then from simple propagation of errors, the noise in each of the H-transform coefficients is also σ. To compress noisy images, divide each coefficient by $S\sigma$, where $S \sim 1$ is chosen according to how much loss is acceptable. This reduces the noise in the transform to $0.5/S$, so that large portions of the transform are zero (or nearly zero) and the transform is highly compressible.

Why is this better than simply thresholding the original image? As discussed above, if we simply divide the image by σ then we lose all information on objects that are within 1σ of sky in a *single* pixel, but that are detectable by averaging a *block* of pixels. On the other hand, in dividing the H-transform by σ, we preserve the information on any object that is detectable by summing a block of pixels! The quantized H-transform preserves the mean of the image for every block of pixels having a mean significantly different than that of neighboring blocks of pixels.

As an example, Figure 1 shows a 128×128 section (3.6×3.6 arcmin) from a digitized version of the Palomar Observatory–National Geographic Society Sky Survey plate containing the Coma cluster of galaxies. Figures 2, 3, and 4 show the resulting image for $S \simeq 0.5$, 1, and 2. These images are compressed by factors of 10, 20, and 60 using the coding scheme described below. In all cases a logarithmic gray scale is used to show the maximum detail in the image near the sky background level; the noise is clearly visible in Figure 1. The image compressed by a factor of 10 is hardly distinguishable from the original. In quantizing the H-transform we have adaptively filtered the original image by discarding information on some scales and keeping information on other scales. This adaptive filtering is most apparent for high compression factors (Fig. 4), where the sky has been smoothed over large areas while the images of stars have hardly been affected.

The adaptive filtering is, in itself, of considerable interest as an analytical tool for images (Capaccioli et al. 1988). For example, one can use the adaptive smoothing of the H-transform to smooth the sky without affecting objects detected above the (locally determined) sky; then an accurate sky value can be determined by reference to any nearby pixel.

The blockiness that is visible in Figure 4 is the result of difference coefficients being set to zero over large areas, so that blocks of pixels are replaced by their averages. It is possible to eliminate the blocks by an appropriate filtering of the image. A simple but effective filter can be derived by simply adjusting the H-transform coefficients as the transform is inverted to produce a smooth image; as long as changes in the coefficients are limited to $\pm S\sigma/2$, the resulting image will still be consistent with the thresholded H-transform.

4. Efficient Coding

The quantized H-transform has a rather peculiar structure. Not only are large areas of the transform image zero, but the non-zero values are strongly concentrated in the lower-order coefficients. The best approach we have found to code the coefficient values efficiently is quadtree coding of each bitplane of the transform array. Quadtree coding has been used for many purposes (see Samet 1984 for a review); the particular form we are using was suggested by Huang and Bijaoui (1991) for image compression.

- Divide the bitplane up into 4 quadrants. For each quadrant code a '1' if there are any 1-bits in the quadrant, else code a '0'.
- Subdivide each quadrant that is not all zero into 4 more pieces and code them similarly. Continue until one is down to the level of individual pixels.

This coding (which Huang and Bijauoi call "hierarchic 4-bit one" coding) is obviously very well suited to the H-transform image because successively lower orders of the H-transform coefficients are located in successively divided quadrants of the image.

We follow the quadtree coding with a fixed Huffman coding that uses 3 bits for quadtree values that are common (e.g., 0001, 0010, 0100, and 1000) and uses 4 or 5 bits for less common values. This reduces the final compressed file size by about 10% at little computational cost. Slightly better compression can be achieved by following quadtree coding with arithmetic coding (Witten, Bell, and Cleary 1987), but the CPU costs of arithmetic coding are not, in our view, justified for 3–4% better compression. We have also tried using arithmetic coding directly on the H-transform, with various contexts of neighboring pixels, but find it to be both computationally inefficient and not significantly better than quadtree coding.

For completely random bitplanes, quadtree coding can actually use more storage than simply writing the bitplane directly; in that case we just dump the bitplane with no coding.

Note that by coding the transform one bitplane at a time, the compressed data can be viewed as an incremental description of the image. One can initially transmit a crude representation of the image using only the small amount of data that is required for the sparsely populated, most significant bit planes. Then the lower bit planes can be added one by one until the desired accuracy is required. This could be useful, for example, if the data is to be retrieved from a remote database — one could examine the crude version of the image (retrieved very quickly) and abort the transmission of the rest of the data if the image is judged to be uninteresting.

5. Astrometric and Photometric Properties of Compressed Images

Astronomical images are not simply subjected to visual examination, but are also subjected to careful quantitative analysis. For example, for the image in Figure 1 one would typically like to do astrometric (positional) measurements of objects to an accuracy much better than 1 pixel, photometric (brightness) measurements of objects to an accuracy limited only by the detector response and the noise, and accurate measurements of the surface brightness of extended sources.

We have done some experiments to study the degradation of astrometry and photometry on the compressed images compared to the original images (White, Postman, and Lattanzi 1991). Even the most highly compressed images have very good photometric properties for both point sources and extended sources; indeed, photometry of extended objects can be improved by the adaptive filtering of the H-transform (Capaccioli et al. 1988). Astrometry is hardly affected by the compression for modest compression factors (up to about a factor of 20 for our digitized photographic plates), but does begin to degrade for the most highly compressed images.

These results are based on tests carried out with tools optimized for the original images; it is likely the best results will be obtained for highly compressed images only with analysis tools specifically adapted to the peculiar noise characteristics of the compressed images.

6. Conclusions

In order to construct the Guide Star Catalog for use in pointing the Hubble Space Telescope, the Space Telescope Science Institute scanned and digitized wide-field photographic plates covering the entire sky. The digitized plates are of great utility, but to date it has been impossible to distribute the scans because of the massive volume of data involved (a total of about 600 Gbytes). Using the compression techniques described in this paper, we plan to distribute our digital sky survey on CD-ROMs; about 100 CD-ROMs will be required if the survey is compressed by a factor of 10.

The algorithm described in this paper has been shown to be capable of producing highly compressed images that are very faithful to the original. Algorithms designed to work on the original images can give comparable results on object detection, astrometry, and photometry when applied to the images compressed by a factor of 10 or possibly more. Further experiments will determine more precisely just what errors are introduced in the compressed data; it is possible that certain kinds of analysis will give more accurate results on the compressed data than on the original because of the adaptive filtering of the H-transform (Capaccioli et al. 1988).

This compression algorithm can be applied to any image, not just to digitized photographic plates. Experiments on CCD images indicate that lossless compression factors of 3–30 can be achieved depending on the CCD characteristics (e.g., the readout noise). A slightly modified algorithm customized to the noise characteristics of the CCD will do better. This application will be explored in detail in the future.

We gratefully acknowledge grant from NAGW-2166 from the Science Operations Branch of NASA headquarters which supported this work. The Space Telescope Science Institute is operated by AURA with funding from NASA and ESA.

References

Blume, H., and Fand, A. 1989, *SPIE Vol. 1091, Medical Imaging III: Image Capture and Display*, p. 2.

Capaccioli, M., Held, E. V., Lorenz, H., Richter, G. M., and Ziener, R. 1988, *Astronomische Nachrichten*, **309**, 69.

Daubechies, I. 1988, *Comm. Pure and Appl. Math.*, **41**, 909.

Fritze, K., Lange, M., Möstl, G., Oleak, H., and Richter, G. M. 1977, *Astronomische Nachrichten*, **298**, 189.

Haar, A. 1910, *Math. Ann.* **69**, 331.

Huang, L, and Bijaoui, A. 1991, *Experimental Astronomy*, **1**, 311.

Richter, G. M. 1978, *Astronomische Nachrichten*, **299**, 283.

Samet, H. 1984, *ACM Computing Surveys*, **16**, 187.

White, R. L., Postman, M., and Lattanzi, M. 1991, in *Proceedings of the Edinburgh Meeting on Digital Sky Surveys*, in press.

Witten, I. H., Radford, M. N., and Cleary, J. G. 1987, *Communications of the ACM*, **30**, 520.