

# **Another Technique for Compressing Astronomical Imaging**

Michael W. Richmond and Nancy E. Ellman

Department of Astrophysical Sciences, Princeton University, Princeton NJ 08544

richmond@astro.princeton.edu, nan@astro.princeton.edu

Received XX ; accepted XX

## ABSTRACT

We describe a technique which has not previously been applied to astronomical images. The method separates the “signal” portion of every pixel value from the “noise” portion, and attempts to compress the former, but transcribes the latter verbatim. In typical astronomical images, fewer than half the bits in each pixel are truly noisy; this method can therefore provide lossless compression by factors greater than two. We provide standalone software (via WWW) to implement our technique on 16-bit unsigned FITS images.

## 1. Introduction

Since modern imaging devices are capable of producing many Megabytes, or even Gigabytes, of data in a single night, the ability to compress astronomical images has become almost a necessity. Many methods have been proposed by different authors, some suited for special situations, some more general. There are lossless schemes: FITSPRESS (Press 19XX), COMPFITS (Veran & Wright 19XX); lossy schemes: Pyramidal Mean (Starch, Murtaugh & Louys 19XX), integer cosine transform (Haines *et al.* 1994); and some that are both: the H-transform (White & Percival 1994).

In the course of developing software for the Sloan Digital Sky Survey (SDSS; Kent 1994; Gunn 1995), we have uncovered a method of data compression which is not widely known in the astronomical community, although long used by computer scientists.

We describe a general-purpose algorithm which uses this technique as a service to others who may use it as a starting point for further work. It is possible that the SDSS might distribute compressed images in this format, in which case this paper might serve as a primer for those who may need to write their own decoding programs. We provide a WWW site,

<http://www.astro.princeton.edu/richmond/rice/rice.html>

at which readers can find ANSI-C programs which implement this algorithm in a simple way.

## 2. The Algorithm

The basic idea of our method is to split each pixel value into two sections: the high-order bits, which (usually) change very little from pixel to pixel, and the low-order “noisy” bits, which change substantially. Several sources can cause values to differ slightly between neighboring pixels: readout noise, Poisson variations in photons arriving from the sky, flatfield variations, etc. In many CCD cameras, the gain, or conversion factor between electrons and analog-to-data units (ADUs), is set so that typical exposures lead to pixel-to-pixel variations in the lowest 3 to 5 bits. This leaves the bulk of each pixel value (13 to 11 bits, for a 16-bit device) free to record significant features.

We consider only lossless compression, in which the uncompressed image is a perfect replica of the original. Since the “noisy” sections of the data vary quasi-randomly from pixel to pixel, we are unable to devise a set of rules that can reproduce them which are smaller than the data themselves. However, the high-order sections of each pixel value *can* be encoded, because their many bits almost always contain the same pattern.

Let us use an example to illustrate this point. Imagine a set of 4 consecutive pixel values from some section of an image containing “blank sky,” with a sky value approximately 3000. In the first two rows of Table 1, we write their values in decimal and in 16-bit binary notation. Note that each of the four binary numbers starts with the same string of 12 digits: 000010111011. The only differences between the binary values appears in the final 4 bits. One can imagine several schemes that would somehow encode the 12-digit string once, then list explicitly only the lowest-order 4 bits for each pixel. Such encoding would reduce the number of bits needed to represent this set of 4 pixels from  $16 * 4 = 64$  bits to approximately  $12 + 4 * 4 = 28$  bits, yielding a compressed dataset only  $28/64 = 44\%$  the size of the original.

One important input to our algorithm is the number of “noisy” bits present in an image. It is only necessary to gauge this number to a bit better than a factor of 2 or so. A

simple calculation of the standard deviation of pixel values will probably yield too large a value, since it will be unduly influenced by the few pixels with high values in and around bright objects. More robust measures of the distribution of pixel values – clipped mean, interquartile range, or the width of a Gaussian fit to the histogram of intensities – provide an adequate indication of the noise level. For example, suppose that a Gaussian fit to the histogram of pixel intensities in an image has a width  $\sigma = 20$  counts. Neglecting the tiny number of pixels with high values, roughly 66% of all pixels have values within  $\pm 1\sigma = \pm 20$  counts of the peak of the histogram, and about 95% of all pixels have values within  $\pm 2\sigma = \pm 40$  counts of the peak. One might conclude that the pixels in that image have about  $\log_2 80 \approx 6$  bits of noise, and therefore  $16 - 6 = 10$  bits which might be compressed with profit.

Our method is based upon this notion of separating all pixel values into “signal” and “noise”, but with a couple of minor variations. First, we “delta-encode” the original data, calculating the difference between each pixel value and its neighbor. This serves to set most of the highest-order bits to 0, which will be useful in a later step. Unfortunately, it also forces the very highest-order bit, the sign bit, to become 1 for negative numbers. Since this would cause the pattern of high-order bits to differ very frequently, we perform a little trick: given pixel values  $p_1$  and  $p_2$ , we define an unsigned quantity

$$\Delta = \begin{cases} 2 * (p_1 - p_2), & \text{if } p_1 \geq p_2; \\ 2 * (p_2 - p_1) - 1, & \text{if } p_1 < p_2. \end{cases}$$

It is, of course, necessary to place the very first pixel value into an encoded stream; but all subsequent values can be replaced by this  $\Delta$ . Using the pixels listed in the first row of Table 1, converting to  $\Delta$  yields the values shown in rows 3, 4, and 5 of Table 1. Note that all encoded values, except the first, consist of long strings of consecutive 0-bits.

Given this delta-encoded stream of pixel values, we next must choose a manner by which we can compress the long strings of nearly-identical high-order bits. We start with

the number  $N$  of “noisy” low-order bits in an image, and its complement  $M = 16 - N$ , the number of “signal-filled” high-order bits. In the example above, we could select  $N = 4$ , and so  $M = 12$ . Let us consider the highest-order  $M$  bits in the second number, for which  $\Delta = 16$ . Ignoring the lowest-order  $N$  bits, we find a binary number  $K_{\text{binary}} = 000000000001$ , which can be converted to the decimal value  $K_{\text{decimal}} = 1$ . The highest-order  $M$  bits of the third and fourth pixels’  $\Delta$  values, 11 and 8, both yield the decimal value  $K_{\text{decimal}} = 0$ . If one chooses  $M$  wisely, then the set of  $M$  high-order bits should almost always yield a very small number. We can encode this number in a simple fashion by creating a string of  $K_{\text{decimal}}$  bits set to zero. Thus, for the second pixel, we would create a single zero: 0. For the third and fourth pixels, we would create a set of zero zeroes: . Each set is clearly more compact than the  $M = 12$  bits needed to represent these numbers in the original data.

However, we face a problem when a delta-encoded pixel has an unusually high value. The highest  $M = 12$  bits in first pixel in our example above, which must encode the value  $K_{\text{binary}} = 000010111011$ , would yield a set of  $K_{\text{decimal}} = 187$  zeroes, which is many more than the original  $M = 12$  bits. In this case, it would seem that our encoding scheme inflates the data volume, rather than compressing it. At some point, therefore, it is better to represent pixel values in an un-encoded fashion. Let us define a number  $K_{\text{max}}$  which represents the maximum value the upper  $M$  bits can have and still be encoded profitably. We test each delta-encoded value before encoding it: if the top  $M$  bits represent a number sufficiently large that it would yield a set of more than  $K_{\text{max}}$  zeroes, we do *not* encode it.

How can we determine  $K_{\text{max}}$ ? Let us add up the number of bits it takes to encode a (delta) pixel value: we use  $K_{\text{decimal}}$  bits, each set to 0, for the high-order bits, plus  $N$  bits for the low-order, “noisy” portion. We allow at most  $K_{\text{max}}$  consecutive 0-bits. If we choose *not* to encode a (delta) pixel value, we can mark that fact by placing  $K_{\text{max}} + 1$  consecutive 0-bits at the start of a number, then append the “natural” 16-bit representation for the (delta) pixel value. Comparing these two expressions, we find that there is no general solution for

$K_{\max}$ . If one underestimates the noise in an image, and so chooses too small a value of  $N$ , many pixels will have three or four bits set in their upper  $M$ , leading to relatively large values of  $K_{\text{decimal}}$  and hence long sets of 0-bits; in this case, a large value of  $K_{\max}$  is needed to accomodate the long strings of 0-bits. On the other hand, if one has chosen a value for  $N$  a bit larger than the optimal one, very few pixels will have any non-zero bits in their upper  $M$ , in which case a value of  $K_{\max}$  of 0 or 1 may provide the best performance. We conclude that the best setting for  $K_{\max}$  might vary considerably, based on the characteristics of the data.

### 3. Step-by-step instructions for writing compressed data

We now provide a succinct description for writing compressed data with our algorithm.

**Step 1.** Choose values  $N$  = number of noise bits per pixel,  $K_{\max}$  = length of longest allowed set of 0-bits. Let  $M = 16 - N$  and  $\delta_0 = 2^{K_{\max}-1}$ . Let  $p$  = the previous encoded pixel value and set  $p = 0$  to start.

**Step 2.** Read  $P$  = the next pixel value from the image.

**Step 3.** Calculate  $\Delta = P - p$ .

**Step 4.** If  $\Delta \geq -\delta_0$  or  $\Delta < \delta_0$ , then goto step 5a. Otherwise, goto step 5b.

**Step 5a.** Encode the value  $\Delta$  as follows: calculate

$$val = \begin{cases} 2 * \Delta, & \text{if } \Delta \geq 0; \\ -2 * \Delta - 1, & \text{if } \Delta < 0. \end{cases}$$

Let  $K$  equal the highest-order  $M$  bits in  $val$ , as if one shifted the bits in  $val$  rightwards  $N$  times. Write  $K$  bits, each set to 0, to the output (even if  $K = 0$ ). Write 1 bit, set to 1, to the output. Write the  $N$  lowest-order bits in  $val$  to the output. Goto step 6.

**Step 5b.** We will not encode the value  $\Delta$ , but will instead write it directly to the output in its full 16-bit form. However, we first create a marker to signal this decision. Write  $K_{\max} + 1$  bits, set to 0, to the output. Write 1 bit, set to 1, to the output. Write the 16-bit value of  $\Delta$  to the output. Goto step 6.

**Step 6.** Let  $p = P$ , and goto step 2.

As an example, we provide in Table 1 the progression of our algorithm on a set of 4 pixel values. We show, in successive rows of the table, raw data values, delta-encoded values, and the string of “output” compressed bit values. In this example, we chose  $N = 4$  “noisy” bits, and  $K_{\max} = 6$ . The length of the original data (row 2 of Table 1) is  $4 * 16 = 64$  bits, while the length of the compressed string (row 6) is 39 bits. Note that more than half of the compressed string serves to encode the initial pixel value; in a long string of similar pixel values (as is typical in astronomical images), the influence of this initial, long encoded value would be overwhelmed by the multitude of compact encoded numbers.

#### 4. Step-by-step instructions for uncompressing data

Given a set of compressed data, which we imagine as a very long string of bits, we can recreate the original data values as follows:

**Step 1.** Choose values  $N =$  number of noise bits per pixel and  $K_{\max} =$  length of longest allowed set of 0-bits. Note that  $N$  and  $K_{\max}$  **must** match the values used to compress the data exactly. Let  $M = 16 - N$  and  $\delta_0 = 2^{K_{\max}-1}$ . Let  $p =$  the previous encoded pixel value and set  $p = 0$  to start.

**Step 2.** Read bits from the input until reaching a bit set to 1. Let  $K =$  the number of 0-bits read. Then skip past the 1-bit which broke the streak of consecutive 0-bits.

**Step 3.** If  $K \leq K_{\max}$ , goto step 4a. If  $K = K_{\max} + 1$ , goto step 4b. If  $K > K_{\max} + 1$ , an error has occurred.

**Step 4a.** Read the next  $N$  bits from the input, and interpret them as an integer  $n$ .

Calculate  $val = 2^N * K + n$ . Goto step 5.

**Step 4b.** Read the next 16 bits from the input, and interpret them as an (unsigned)

integer  $n$ . Calculate  $val = n$ . Goto step 5.

**Step 5.** Check whether  $val$  is an even or odd number, and let

$$\Delta = \begin{cases} val/2, & \text{if } val \text{ is even;} \\ -val/2 + 1, & \text{if } val \text{ is odd.} \end{cases}$$

**Step 6.** Let  $P = p + \Delta$ . Write the pixel value  $P$  to the output.

**Step 7.** Set  $p = P$  and goto step 2.

## 5. Compression and the FITS header

There are two approaches one can take to compressing data which is stored in a FITS file: one can apply a technique to the entire file, including the header (as one might by running **compress** or **gzip**), or one can attempt to leave the FITS header untouched and modify only the data section of the file. We have chosen the second option, largely because it is very convenient to have an ASCII text-like segment at the start of a large image which describes its contents. We emphasize, however, that the current (November 1995) FITS standard does not support any sort of data compression. Therefore, our compressed data files cannot be considered true FITS, and might kindly be described as “psuedo-FITS.” Nonetheless, we find our approach a convenient compromise between utility and legality.

Given a 2-dimensional, 16-bit signed integer “true” FITS image, we create a “psuedo-FITS” compressed file in the following manner. We copy the FITS header (almost) exactly,

but make two small changes. First, we set the SIMPLE keyword's value to F, since this is certainly no longer a standard FITS file. We do *not*, however, modify the values of BITPIX, NAXIS, NAXIS1 or NAXIS2. Naively trusting FITS readers may therefore become confused if they try to read information from the data portion of the file. Second, we insert one new keyword into the header:

```
COMPRESS=          'rice_5_8'
```

The value of this keyword includes three components, separated by underscore characters: the word 'rice', the number of "noisy" bits  $N$  (here 5), and the value of  $K_{\max}$  (here 8) used in the compression process. We pad this header to a multiple of 2880 bytes with 'space' characters.

Next, we write the (very long) string of compressed bits, as described in section 2. Issues of byte-swapping may become important here on machines with different conventions for the most- and least-significant bits in a byte.

Finally, we pad the file to an integral multiple of 2880 bytes with 'null' characters (ASCII value 0). We repeat that the length of this file can *not* be derived from any keywords in the pseudo-FITS header; it is most definitely *not* simply  $NAXIS1 * NAXIS2 * 2$  bytes beyond the header.

We suggest modifying the extension of the file name to one which is not used for standard FITS images. For example, if one gives names like "ngc4496.fts" to normal images, one might give compressed images names like "ngc4496.ftz."

When we un-compress a file and produce a truly standard FITS image, we retain the COMPRESS keyword in the header, but set it to

```
COMPRESS=          'none'
```

We hope that this will not confuse any other programs that might have different schemes for compressing FITS data.

## 6. Performance

The performance of our algorithm is a strong function of the characteristics of its input image. Images with very little noise and a uniform background can be compressed by large factors. Many sources can cause images to resist compression, however: high levels of photon noise (due to bright sky or long exposure times), the presence of many objects (as in a crowded stellar field), significant small-scale pixel-to-pixel variations due to sensitivity variations across a detector, large number of cosmic rays, etc. In the worst case, if every pixel varies from its neighbor by more than  $\delta_0$ , our method will inflate an image by a factor of  $(K_{\max} + 1 + 16)/16$ . In the best case, if the image has a perfect distribution of random noise in the lowest-order  $N$  bits, the number of bits per pixel required in the compressed image will be  $N + 1$ , leading to a compressed file only  $(N + 1)/16$  the size of the original.

To measure the performance of our algorithm on a set of real astronomical images, we have used the suite of images provided in “Test Image Descriptions” (Murtagh & Warmels 1989). For each image, we ran our program a number of times, allowing  $N$  to range from 1 to 8 bits. At each value of  $N$ , we set  $K_{\max} = 128/2^{N-1}$  (i.e. used a fixed value of  $\delta_0 = 128$  for all images). We chose the best and second-best values for  $N$  for each image, and list them in Table 2. We also list the compression ratio, defined as the total size of the compressed image (including the FITS header) divided by the total size of the original image (including its FITS header). Note that the compression ratios vary from 26% to 60%, with most values lying between 30% and 40%.

As we mention in Section 2, the optimum value for  $N$  should be closely related to the standard deviation of pixel values in an image. To check this relationship, we created

a histogram of pixel values for each image in the test suite, and fit a Gaussian to the histogram's values near its peak. We list the width  $\sigma$  of the Gaussian in Table 2. As expected, images with low values of  $\sigma$  are best compressed with small  $N$ , and larger  $\sigma$  require larger  $N$ .

One image, however, breaks this pattern: com0001, a section of the halo of a giant elliptical galaxy in the Coma cluster, requires  $N = 6$ , larger than any other image with  $\sigma < 20$ . Closer examination of this image revealed that the distribution of pixel values is unusual. The histogram of pixel values shows strong, repeating cycles between a value  $X$ ,  $0.5X$  and 0. For example, there are 814, 1621, 805, 813, 0, 1619, 800, 0, 816, 1647, 802 pixels with values of 1639, 1640, ... , 1649. It appears that either the analog-to-digital converter misbehaved when this image was taken, or that the image was somehow corrupted during the reduction or storage process. Perhaps it is only the copy at the FTP site <ftp://iraf.noao.edu/iraf/extern/focas.std.tar.Z> which exhibits this behavior. If one ignores this spurious variation in the histogram of pixel values, one finds it may be described roughly as a Gaussian with  $\sigma \approx 80$  counts. One can also calculate the standard deviation from the mean inside small boxes at random points in the image, and find  $\text{stdev} \approx 40$  counts. It is then clear that the large value for  $N$  is justified.

## 7. Free Software

We have written a simple, standalone program that can perform compression and decompression of FITS images using the method we have described above. Our intent is to provide a portable code that can be used on many different machines. Therefore, we have used strict ANSI C throughout, and resisted the temptation to use one of several fine FITS I/O packages, such as FITSIO (Pence 1993), in order to reduce the amount of software that must be transferred, compiled and often (sadly) modified to fit a user's particular

computing system.

Interested readers can get a copy of this software, or read a more detailed description, by contacting the first author or accessing the WWW site

<http://www.astro.princeton.edu/richmond/rice/rice.html>

## REFERENCES

Gunn, J. E. 1995, BAAS, 186, 4405

Haines, R. F., Gold, Y., Grant, T., & Chuang, S. 1994, NASA report N95-11004

Kent, S. M. 1994, Ap&SS, 217, 27

Murtagh, F., & Warmels, R. H. 1989, in Proceedings of 1st ESO/ST-ECF Data Analysis Workshop, ed. P. J. Grosbol, F. Murtaugh, & R. H. Warmels, (Garching: ESO)

Pence, W. 1993, BAAS, 182, 1603

White, R. L. & Percival, J. W. 1994, BAAS, 185, 11504

Table 1. Example of stages in compression

Stage	Pixel 1	Pixel 2	Pixel 3	Pixel 4
raw value	2995	3003	2997	3001
raw, binary	0000101110110011	0000101110111011	0000101110110101	0000101110111001
difference	2995	8	-6	4
unsigned $\Delta$	2995 <sup>a</sup>	16	11	8
$\Delta$ , binary	0000101110110011	000000000010000	000000000001011	000000000001000
output	000000010000101110110011	010000	11011	11000

<sup>a</sup>Since  $2995 > \delta_0$ , it is not converted into unsigned  $\Delta$ .

Table 2. Performance on sample images<sup>a</sup>

Image	Object	Sky $\sigma$	Best N, ratio	2 <sup>nd</sup> -best N, ratio	Comments
for0001	Fornax dwarf	4.88	3, 0.33	4, 0.34	crowded star field
for0002	Fornax dwarf	2.88	2, 0.26	3, 0.27	crowded star field
com0001	halo of NGC 4874	9.01 <sup>c</sup>	6, 0.50	7, 0.52	strong gradient across field
gal0001	galaxy cluster	8.50	4, 0.37	5, 0.39	single exposure
gal0002	galaxy cluster	19.04	5, 0.45	6, 0.47	sum of six exposures
gal0003 <sup>b</sup>	SA 68	3.96	3, 0.32	4, 0.33	“blank” field
gal0004 <sup>b</sup>	SA 68	4.93	3, 0.32	4, 0.33	another “blank field
ngc0001	NGC 3201	21.10	6, 0.60	7, 0.62	glob. cluster, med. density
ngc0002	NGC 3201	4.19	4, 0.45	5, 0.46	shorter exposure
sgp0001 <sup>b</sup>	deep SGP in J	4.25	3, 0.34	4, 0.34	very deep “blank” field
sgp0002 <sup>b</sup>	deep SGP in R	4.20	3, 0.30	4, 0.32	very deep “blank” field
tuc0003	47 Tuc off-center	5.20	4, 0.39	5, 0.42	not too crowded
tuc0004	47 Tuc off-center	3.62	4, 0.38	5, 0.41	shorter exposure

<sup>a</sup>See Murtagh & Warmels 1989.

<sup>b</sup>Converted image from 32-bit floating point to 16-bit integer before starting test.

<sup>c</sup>Actual standard deviation from mean  $\sim 40$  counts. See text.