User's Guide

Fpack FITS Image Compression Utility

W. D. Pence, NASA/GSFC
R. Seaman, NOAO
R. L. White, STScI

January 2010

1. Introduction

Fpack is a utility program for optimally compressing images in the FITS (Flexible Image Transport System; http:// fits.gsfc.nasa.gov) data format. The associated *funpack* program restores the compressed image file back to its original state. These programs may be run from the host operating system command line and are analogous to the GZIP and GUNZIP utility programs except that they are optimized for FITS format images and offer a wider choice of compression options.

Fpack stores the compressed image using the FITS tiled image compression convention (http:// fits.gsfc.nasa.gov/fits_registry.html). Under this convention the image is first divided into a user-configurable grid of rectangular tiles, and then each tile is individually compressed and stored in a variable-length array column in a FITS binary table. By default, *fpack* usually adopts a row-by-row tiling pattern.

The tiled image compression convention can support any number of different compression algorithms. The *fpack* and *funpack* utilities call on routines in the CFITSIO library (http:// heasarc.gsfc.nasa.gov/fitsio) to perform the actual compression and uncompression of the FITS images. Currently, the GZIP, Rice, and H-compress general purpose algorithms and a more specialized PLIO IRAF pixel list compression algorithm are supported.

2. Image Compression Overview

This section provides background information about image compression in general and the FITS tiled image convention in particular. This is followed in Sections 3 and 4 by more detailed information on the *fpack* and *funpack* command line options.

2.1. Benefits of *fpack* and the Tiled Image Compression Algorithm

Using *fpack* to compress FITS images offers a number of advantages over the other commonly used technique of externally compressing the whole FITS file with GZIP:

1. *Fpack* generally offers higher compression ratios and faster compression speed than GZIP.
2. The FITS image header keywords remain uncompressed and can be read or written without any additional overhead.
3. Each HDU of a multi-extension FITS file is compressed separately, thus it is not necessary to uncompress the entire file to read a single image in a multi-extension file.
4. The feature of dividing the image into rectangular tiles before compression enables faster access to small subsections of the image because only those tiles contained in the subsection need be uncompressed.
5. The compressed image is itself a valid FITS file and can be manipulated by other general FITS utility software.
6. *Fpack* also supports lossy compression techniques that achieve significantly higher compression than lossless compression algorithms in situations where it is not necessary to exactly preserve every bit of the original image pixel values. This is especially relevant when compressing 32-bit floating point FITS images where there is usually little justification for preserving the full 6 decimal places of numerical precision of each pixel value.
7. *Fpack* and *funpack* automatically update the CHECKSUM keywords in the compressed and uncompressed files to help verify the integrity of the FITS files.

Data providers can minimize the data storage and network bandwidth resources needed to archive and distribute FITS images by compressing them first with *fpack*. Users can then uncompress the file with *funpack* to convert them back into the standard FITS image format before doing further analysis. The benefits of using *fpack* are magnified, however, when the analysis software is capable of directly reading and writing the files in the compressed form.

Any software application that uses the CFITSIO library (http://heasarc.gsfc.nasa.gov/fitsio) will inherit the ability to read or write tile-compressed FITS images. The image compression or uncompression is performed internally by the CFITSIO library routines, so in general, the application program itself does not need to know anything about the tiled image compression format. The main exception is that when writing compressed images, the application program may need to call an additional routine to define which compression algorithm to use, along with the values of other optional compression parameters. The *fpack* and *funpack* utilities are themselves examples of applications that use CFITSIO to perform the compression and uncompression operations on the images.

In addition to CFITSIO, the ds9 image display program and the IRAF data analysis system currently provide some support for the tile-compressed FITS image format. It is anticipated that other analysis systems will also add support for this tiled image compression format as it become more widely used. In the meantime, *funpack* may be used to uncompress the images back into standard FITS images for compatibility with other analysis software that does not yet directly support the compressed format.

2.2. Compression Versus Noise

When images are losslessly compressed, the compression ratio depends almost completely on one simple factor: the amount of the noise in the pixel values. The noise, by definition, cannot be compressed, so the compression ratio of an image will be inversely proportional to the total number of noise bits in the image. As is discussed in greater detail in a separate paper (Pence, Seaman, & White, 2009; PASP 121,414; http://arxiv.org/abs/0903.21401), the amount of noise in a image can be calculated from the measured standard deviation (σ) of the pixels in the "background" areas of the image (e.g., excluding bright stars or other objects in the image) which typically have close to a Gaussian intensity distribution. The average number of noise bits per pixel is given by

$$N_{bits} = log_2(\sigma\sqrt{12}) = log_2(\sigma) + 1.792 \quad (1)$$

Since these noise bits are fundamentally uncompressible, the maximum possible compression ratio, in the ideal case where all the remaining bits are compressed to zero, is simply given by the ratio BITPIX / N_{bits} (where BITPIX is the number of bits in each pixel value). No actual compression algorithm can achieve this theoretical limit, so in practice the compression ratio is given by

$$R = BITPIX/(N_{bits} + K) \quad (2)$$

where K is a measure of the efficiency of the particular compression algorithm. For the Rice algorithm, K has a value of about 1.2, and for Hcompress it is about 0.8. The k value for GZIP is much larger, typically about 4 or 5.

2.3. Lossless Compression of Integer FITS Images

In the previously mentioned paper, we used a large set of direct imaging CCD exposures of star fields in the night sky, plus the associated calibration exposures, to compare the compression speeds and file compression ratios for the 3 general purpose compression algorithms that are currently supported by *fpack*, namely, Rice, GZIP, and Hcompress. We also compared these to the method of compressing the entire FITS file with the host-level GZIP file compression program.

Figure 1 neatly summarizes the main results of that study. It shows that the Rice and GZIP compression ratios are tightly correlated with the amount of noise in the image as predicted by

equation 2 (shown by the solid lines). The Rice algorithm clearly produces much better compression than GZIP, and is within 85% – 90% of the maximum possible compression (for an algorithm with $K = 0$) show by the dashed line. It is also interesting to note that the different types of images – bias frames, short calibration exposures of reference stars, and deep exposures of the sky – contain distinctly different amounts of noise because the photon shot noise is proportional to the square root of the number of detected photons

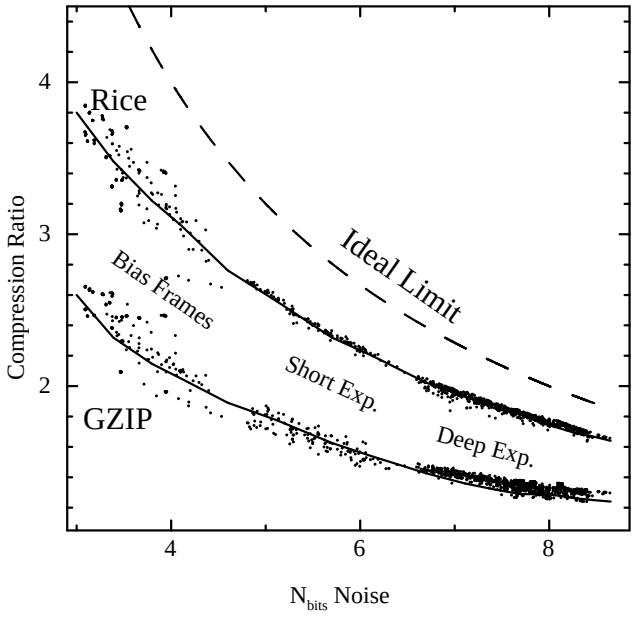

Fig. 1.— Comparison of the compression ratios using Rice and GZIP plotted as a function of the noise in the 16-bit integer images. The different types of images contain characteristically different amounts of noise, as labeled. The dashed line shows the theoretical upper limit for an ideal compression algorithm.

The mean file compression ratios and the relative compression and uncompression CPU times for these 4 different compression methods are listed in Table 1. These values are the mean for all 16-bit integer images in the sample data set and the CPU times are relative to the speed of the Rice algorithm.

As shown in the table, the Rice and Hcompress achieve significantly larger compression of these astronomical images than GZIP. The GZIP compressed files are on average about 1.4 times larger than the Rice or Hcompressed files. Hcompress produces about 3% better compression than Rice, but for most applications this small gain is not worth the much greater CPU times required to compress and uncompress the images with Hcompress.

The Rice compression algorithm is much faster than Hcompress. Rice compression is also much faster than GZIP and the uncompression speed is about the same as GUNZIP. Note that the

factor of 2 difference in speed between the host-level GZIP program and the identical algorithm used within the tiled FITS image implementation is due to the difference in I/O methods. The host-level GZIP program can read and write the files more efficiently as continuous streams, whereas the FITS implementation requires that the input and output files be copied to and from intermediate storage buffers to provide random access to any location within the FITS file. As a benchmark point of reference, a Linux machine with a 2.4 GHz AMD Opteron 250 dual core processor can compress or uncompress a 50 MB 16-bit integer image (5000 x 5000 pixels) with the Rice compression algorithm 1 second of CPU time .

Similar trends are seen when compressing 32-bit integer images, only the compression factors that are achieved are typically twice that of a 16-bit image, given the same noise level in both images.

2.4. Compression of Floating Point FITS Images

It is generally not practical, nor necessary, to losslessly compress floating point FITS images (that have BITPIX = -32 or -64). This is because most of the algorithms do not support compression of floating point data, and even when they do (e.g., GZIP), there is rarely any justification for preserving the full 6 decimal places of precision in each pixel value. In practice, a large fraction of the bits in the mantissa of each pixel value only contain uncompressible noise. For this reason, *fpack* usually first converts the floating point pixel values into 32-bit integers using a linear scaling function:

$$IntegerValue = (FloatingPointValue - ZeroPoint)/ScaleFactor \quad (3)$$

This array of scaled integers is then compressed with the Rice algorithm by default. When the image is subsequently uncompressed, the integer values are inverse scaled to closely, but not exactly, reproduce the original floating point pixel values. Separate scale and zero point values are computed for each tile of the image.

The value of ScaleFactor controls how closely the inverse scaled values approximate the original floating point values. Making ScaleFactor smaller reduces the spacing between the quantized levels and thus more closely preserves the original pixel values. However, this also magnifies the dynamic

Table 1. 16-bit integer image compression

	Rice	Hcompress	Tiled-GZIP	Host-GZIP
Compression Ratio	2.11	2.18	1.53	1.64
Relative Compression Time	1.0	2.8	5.6	2.6
Relative Uncompression Time	1.0	3.1	1.9	0.85

range and the noise level in the integer array that is to be compressed which adversely affects the amount of compression that is achieved. Thus, there is a direct trade-off between providing greater fidelity or achieving greater compression.

Since it is not easy to directly determine an appropriate ScaleFactor value for a given image, *fpack* instead provides a parameter called "q" for specifying the quantized level spacing relative to the sigma of the measured noise in background areas in the image. The image pixel values will be quantized into levels that have a spacing of sigma/q. The number of noise bits that are retained in each pixel value is given by

$$N_{bits} = log_2(q) + 1.792 \tag{4}$$

From this one can compute the expected compression ratio as a function of q from Equation 1.

One important refinement to this quantization procedure is to add a small amount of random noise to the floating point value before scaling it to an integer. That same random value is subtracted when converting back to the floating point value, so there is no net increase in the amount of noise in the image. This has the effect of randomly shifting the zero point in the grid of quantized levels for each pixel, so that the whole image is not quantized into exactly the same grid of levels. This technique, called "subtractive dithering", makes it possible to detect low amplitude features in the quantized image through an effect known as "stochastic resonance". In astronomical images this is especially important for accurately preserving the mean value of the background sky level which is needed when measuring the flux of faint sources in the image.

Using too small a value of q could result in unrecoverable loss of information, whereas too large a value of q will needlessly preserve more of the noise in the image and reduce the amount of compression that is achieved. Recent tests (Pence, White, & Seaman, ADASS 2009) suggest that the default value of q = 4 in *fpack* is quite conservative and that smaller values of q as low 1 may be sufficient to preserve the significant information in typical astronomical images. Users are urged to perform quantitative tests on there own data sets using different values of q to determine the appropriate value for their particular application.

In rare situations, or for test purposes, it may be desirable to losslessly compress floating point FITS images. This can be accomplished in *fpack* by selecting the GZIP compression algorithm and a q value of 0 (e. g., "fpack -g -q 0"). This will preserve every bit in the floating-point pixels at the expense of much lower compression ratios than if the recommended integer quantization technique is used.

3. *fpack* Command Line Options

The *fpack* program is invoked on the computer operating system command line,

```
fpack [Options] [FileNames]
```

where the "Options" control the various compression options and "FileNames" is a list of one or more FITS file names to be compressed. The available options are described below and must appear before the list of files. The file names may contain the usual wildcard characters (*, $, etc.) that will be expanded by the command shell. An input FITS file can be read from the standard stdin file stream by specifying a hyphen as the file name.

1. **Compression Algorithm Options**

```
-r     Rice  [default]
-h     Hcompress
-g     GZIP (per-tile)
-p     IRAF pixel list compression algorithm.  This can only be applied
         to images whose pixel values all lie in the range 0 to 2**24.
-d     No compression (debugging mode)
```

2. **Tiling Pattern Options**

   A row by row tiling pattern is used by default with the Rice, GZIP, and PLIO compression algorithms (i.e., the tiles are one dimensional and contain NAXIS1 pixels each). The Hcompress algorithm is inherently 2-dimensional, therefore the default is to use 16 rows of the image per tile. If this would cause the last tile of the image to only contain a small number of rows, then a slightly different tile size is chosen so that the last tile is similar in size to the other tiles. The default tile sizes can be overridden with one of the following *fpack* options:

```
-w         Compress the whole image as a single large tile.  Appropriate
             for small files where the rows are too short to be
     efficiently compressed individually.
-t <axes>  Comma separated list of tile dimensions (e.g., -t 200,200
             for tiles that are 200 x 200 pixels in size)
```

3. **Floating Point Image Compression Options**

```
-q <level>   Quantized level spacing [Default = 4]
```

The pixel values in floating point FITS images are quantized into linearly scaled integer values prior to being compressed with the Rice algorithm. This improves the compression ratio by eliminating some of the noise in the pixel values, but consequently the original pixel values are not exactly preserved. The fidelity of the compressed image is controlled by specifying how finely the quantized levels should be spaced relative to the sigma of the noise measured in the background regions of each image tile. The default q value is 4 so that the quantized levels are spaced at 1/4th of the noise sigma value. Recent experiments suggest that this default q value is fairly conservative and that greater compression can be achieved without losing any significant astrometric and photometric precision in the image by using smaller values of q (as low as 2 or 1). The approximate compression ratio for different q values is shown in Table 2.

In some instances it may be desirable to specify the exact q value (not relative to the measured noise), so that all the tiles in the image are compressed using the identical value. This is done by specifying the negative of the desired value. The -T option (described below) can be used to calculate the noise level in the image, which may be useful in determining an appropriate q value.

A numerical technique called "subtractive dithering' is applied to the quantized values to better preserve faint features in the image (as described previously). For better efficiency, *fpack* uses one of 10000 pre-established sequences of random numbers when doing this dithering. It is undesirable to use exactly the same dithering sequence for every image because this can cause artifacts in the difference or sum of 2 compressed images. Which sequence to use for a given image is determined by a 'seed' value that is computed in one of 4 ways:

(a) By default, the seed is computed from the system clock time when the program starts. This ensures that a different seed is randomly chosen each time *fpack* is run, but it also means that the pixel values in the compressed image will be slightly (but insignificantly) different each time it is compressed with *fpack*.

(b) If one wants the same seed to be reused every time a given image is compressed, then specify "-qt" instead of "-q". In this case the seed value will be computed from the checksum of the first tile of image pixels that is compressed. The same seed value will be used for a given image, however, if 2 images have the same pixel values in the first tile (e.g., if there is a blank border around both

Table 2. Floating Point Image Compression Ratios

q	16	8	4	2	1
Ratio	4.5	5.4	6.5	8.2	11

images) then both images will be compressed using the same dithering pattern.

(c) One can specify exactly which one of the 10000 dithering patterns to use by appending an integer number in the range 1 to 10000 to q, as in "-q3008".

(d) Lastly, one can turn off subtractive dithering by specifying -q0. This option is not recommended for general use.

`-n <sigma>`

This rarely used parameter rescales the pixel values in a previously scaled image to improve the compression ratio by reducing the noise in the image. This option is intended for use with FITS images that use scaled integers to represent floating point pixel values, and in which the scaling was chosen so that the range of the scaled integer values covers the entire allowed range for that integer data type. The amplitude of the noise in these scaled integer images is typically so huge that they cannot be effectively compressed. This 'n' option rescales the pixel values so that the noise sigma will be equal to the specified value of n. Appropriate values of sigma will likely be in the range from 1 (for low precision and the high compression) to 16 (for the high precision and lower compression).

4. **Lossy compression of integer images with Hcompress**

`-s <scale>`

Scale factor for lossy compression when using Hcompress. The default value is 0 which implies lossless compression. Positive scale values are interpreted as relative to the sigma of the noise in the image. Scale values of 1.0, 4.0, and 10.0 will typically produce compression factors of about 4, 10, and 25, respectively, when applied to 16-bit integer images. In some instances it may be desirable to specify the exact scale value (not relative to the measured noise), so that all the tiles in the image, and all the images in a data set, are compressed with the identical scale value, regardless of slight variations in the measure. This is done by specifying the negative of the desired value.

Users should carefully evaluate the compressed images when using this lossy compression option to make sure that any essential information in the image has not been lost.

5. **Output File Name Options**

The compressed output file name is usually constructed by appending '.fz' to the input file name, and the input file is not deleted. This behavior may be modified with the following options:

```
-F     Force the input file to be overwritten by the compressed file
           with the same name.
-D     Delete the input file after creating the compressed output file.
-Y     Suppress the prompts to confirm the -F or -D options
-S     Write the compressed FITS file to the stdout stream instead of to a file.
```

6. **Other Miscellaneous Options**

```
-v    Verbose mode; list each file as it is processed
-L    List information about all the extensions in the input files without compressing the
-C    Do not update the FITS checksum keywords
-H    Display a summary help file that describes the available fpack options
-V    Display the fpack and CFITSIO version numbers
-R <filename>   Write the comparison test report (produced by -T) to a
        text file
-T    Produce a report showing the compression statistics for the
        main compression algorithms.   The input files remain unchanged.
        See the appendix for a description of the report format.
```

4. *funpack* Command Line Options

funpack shares many of the same options as *fpack* as shown below:

1. **Output File Name Options**

   The uncompressed output file name is usually constructed by stripping the '.fz' suffix from the input file name, and the input file is not deleted. This behavior may be modified with the following options:

```
-F         Force the input file to be overwritten by the uncompressed file
             with the same name.
-D         Delete the input compressed file after uncompressing it
-P <pre>   Create the output file name by prepend the <pre> string to the
             input file name
-O <name>  Specify the full name of the uncompressed output file
-S         Write the uncompressed FITS file to the stdout stream
-Z         Recompress the output file with the host GZIP program
```

2. **Other Miscellaneous Options**

```
-v    Verbose mode; list each file as it is processed
-L    List all the extensions in the input files without uncompressing them
-C    Do not update the FITS checksum keywords
-H    Display a summary help file that describes the available funpack options
-V    Display the funpack and CFITSIO version numbers
```

5. Installing *fpack* and *funpack*

The latest pre-built binary executable versions of *fpack* and *funpack* for many common computer platforms are available at http:// heasarc.gsfc.nasa.gov/fitsio/*fpack*. The source code is also include in the source file distribution of the CFITSIO library.

To build *fpack* and *funpack* from the source code on unix systems, first unpack the CFITSIO distribution.tar file into an empty directory, and then execute the following commands:

```
./configure
 make
 make fpack
 make funpack
```

This will create the *fpack* and *funpack* executable files which may be copied to any other suitable location.

A. Example Comparison Report

The comparison report that is produced by the fpack -T option has the following format:

```
-------------------------------------------------------------------------------

File: ct655046.fits
Ext BITPIX Dimens. Nulls    Min    Max     Mean   Sigma  Noise3 Nbits  MaxR
  0  16  (1112,4096)   0 -31503  25967 -26679.3 2.5e+03    56.8   7.6  2.10

  Type   Ratio       Size (MB)     Pk (Sec) UnPk Exact ElpN CPUN  Elp1  CPU1
  Native                                          0.024 0.016 0.013 0.010
  RICE    1.83    9.11 ->   4.98    0.57    0.55 Yes 0.053 0.047 0.045 0.040
  HCOMP   1.85    9.11 ->   4.92    1.91    1.56 Yes 0.175 0.159 0.179 0.162
  GZIP    1.35    9.11 ->   6.73    3.07    1.09 Yes 0.114 0.106 0.108 0.101
  NONE    0.99    9.11 ->   9.18    0.35    0.31 Yes 0.022 0.021 0.015 0.013
-------------------------------------------------------------------------------
```

The parameters given on the 3rd line of the report are:

```
Ext - extension number within the file (zero based)
BITPIX - FITS data type of the image (8, 16, 32, -32 or -64)
Dimens - image dimensions
Nulls - number of undefined or null pixels in the image
Min, Max - the minimum and maximum values in the image
Mean - mean value of all the non-null pixels
Sigma - standard deviation of all the non-null pixels
Noise3 - a measure of the noise in the background regions of the image
Nbits - number of noise bits per pixel = log2(noise3) + 1.792
MaxR - theoretical maximum possible compression ratio = BITPIX / Nbits
```

The table, starting on line 5 of the report has the following columns:

```
Type - name of compression method, if any
Ratio - file compression ratio
Size - uncompressed and compressed sizes of the files, in MB
Pk - the CPU time in seconds to compress the image with fpack
UnPk - the CPU time in seconds to uncompress the image with funpack
Exact - is the compression lossless?
ElpN - elapsed time to read the whole image at one time (s/MB)
```

```
CPUN - CPU time to read the whole image at one time (s/MB)
Elp1 - elapsed time to read the image, one row at a time (s/MB)
CPU1 - CPU time to read the image, one row at a time (s/MB)
```

The first row in this table shows the speed when reading the uncompressed FITS image. The next 3 rows show the compression ratios and speeds when using Rice, Hcompress, and GZIP. The last row shows the speeds when the image is written to and read from the FITS tiled image format without doing any compression.