Fpack and Funpack User's Guide

FITS Image Compression Utilities

William Pence, NASA Goddard Space Flight Center, Greenbelt, MD 20771
Rob Seaman, National Optical Astronomy Observatory, Tucson, AZ 85719
Rick White, Space Telescope Science Institute, Baltimore, MD 21218

23 March 2009

1. Introduction

Fpack is a utility program for optimally compressing images in the FITS (Flexible Image Transport System) data format (see http://fits.gsfc.nasa.gov). The associated funpack program restores the compressed image file back to its original state (if a lossless compression algorithm is used). 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 (see 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 FITS image header keywords remain uncompressed for fast access by FITS reading and writing software.

The tiled image compression convention can in principle 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, which currently supports the GZIP, Rice, H-compress, and the PLIO IRAF pixel list compression algorithms.

The fpack and funpack utilities were originally designed and written by Rob Seaman (NOAO). William Pence (NASA) added further enhancements to the utilities and to the image compression algorithms in the underlying CFITSIO library. Rick White (STScI) wrote the code for the Rice and Hcompress algorithms.

2. Benefits of fpack

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 thus 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 capability of dividing the image up into 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 thus can be manipulated by other general FITS utility software.
6. Fpack supports lossy compression techniques that can achieve significantly higher compression factors than the 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 for which there is often little justification for preserving the full numerical precision (6 - 7 decimal places) 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.
8. Software applications that are built on top of FITS access libraries such as CFITSIO, that internally support the tiled image compression technique, are able to directly read and write the FITS images in their compressed form, thus reducing the amount of disk storage space needed by users.

3. General fpack usage guidelines

In its simplest application, fpack can be used to minimize the data storage and network bandwidth resources needed to archive and distribute FITS images. In this scenario, the data producer compresses the FITS image files with fpack before placing them in the data archive; after downloading the compressed files, each end user of the images then runs funpack to convert them back into the standard FITS image format before doing any further analysis.

The benefit of using fpack to compress FITS images is much greater, however, when the user's analysis software is capable of directly reading and writing the files in the compressed form. This reduces both the required amount of local disk space and the time needed to copy the files from one location to another. Directly reading or writing FITS images in the compressed format may require more CPU resources, but this is offset by a gain in I/O performance because fewer bytes of data need to read or written.

Any software application that uses the CFITSIO library (http://heasarc.gsfc.nasa.gov/fitsio) to read and write FITS images will transparently inherit the ability to read or write tile-compressed images. The image compression or uncompression is performed by the CFITSIO library routines, so in general, the application program itself does not need to know anything about the tiled compressed image format. The main exception to this is that when writing compressed images, the application program may need to call an additional CFITSIO 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.

Besides CFITSIO, the IRAF data analysis system also provides some support for the tile-compressed image format. As this format become more widely used in the future, it is anticipated that other analysis systems will also add support for this tiled image compression format. In the meantime, however, users have the option of using funpack to uncompress the images back into standard FITS images that should be compatible with other astronomical image analysis software that does not directly support the compressed format.

4. Compression versus noise

When dealing with lossless compression of images with integer-valued pixels, the amount of compression that will be achieved depends almost completely on one simple factor: the amount of the noise that is present in the pixel values in the image. The noise, by definition, cannot be compressed, so as the amount of noise is reduced, the compression ratio increases dramatically. We discuss this topic in greater detail in a separate paper to be published in PASP. A preprint of this paper is available at http://arxiv.org/abs/0903.2140.

As shown in that paper, the amount of noise in a image can be measured from the standard deviation of the pixels in the "background" areas of the image (e.g., excluding bright stars of other objects in the image). One can then calculate how many bits of each pixel value effectively contain pure noise from this formula:

$$\text{Nbits} = \log_2(\text{standard deviation}) + 1.792$$

These noise bits cannot be compressed, so the maximum possible compression ratio, in the ideal case where all the other non-noise bits are compressed to zero, is given simply by BITPIX / Nbits. No actual compression algorithm can achieve this limit, so in practice the actual compression ratio is given by BITPIX / (Nbits + K) where K is a measure of the efficiency of the compression algorithm. For the Rice algorithm, K has a value of about 1.2. The value for GZIP is much larger, typically about 4 or 5.

In most image detectors used in astronomy, the noise in each pixel value scales with the square root of the number of detected photons. Thus, the amount of noise in the image naturally increases with the mean count rate, or exposure time. The practical implication of this fact is that the different types of exposures taken during a typical astronomical observing session will have distinctly different amounts of noise and hence will compress by differing amounts. Thus, bias frames with low pixel count values will compress better than flat field images that have much large pixel values.

5. Compression of integer FITS images.

In the above mentioned paper, we used a large set of integer CCD images to compare the speed and file compression ratios for the 3 different general purpose compression algorithms that are currently supported by fpack, namely, Rice, GZIP, and Hcompress. We also compared these to the widely-used method of compressing the entire FITS file with the host-level gzip program.

The mean file compression ratios and the relative compression and uncompression elapsed CPU times for these 4 different compression methods are shown in Table 1. These values are the mean for all 1632 16-bit integer images in the sample data set, and the CPU times in each case are relative to those when using the Rice algorithm.

Table 1. Compression Statistics for 16-bit Integer Images

	Rice	**Hcompress**	**GZIP**	**Host GZIP**
Compression Ratio	2.11	2.18	1.53	1.6
Relative compression CPU time	1.0	2.8	5.6	2.6
Relative uncompression CPU time	1.0	3.1	1.9	0.9

As shown in the first row in Table 1, the Rice and Hcompress methods achieve much greater 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. This depends slightly on the amount of noise in the image: the ratio is about 1.3 for the images with the most amount of noise and about 1.5 for the least noisy images. Hcompress produces slightly better compression than Rice (about 3% smaller), but for most applications this small gain is not worth the much greater CPU times required to compress and uncompress the images.

The Rice compression algorithm is considerably faster than either Hcompress or gzip. Note that the timing difference between the host-level gzip and the implementation of this same algorithm within fpack/CFITSIO is mainly due to the fact that the host-level gzip program can read and write the files as simple continuous streams, whereas the fpack implementation requires that the input and output files be copied to and from intermediate storage buffers in memory.

When uncompressing the files, the Rice algorithm is 2 or 3 time faster than the gzip or Hcompress methods offered by fpack/funpack and has about the same speed as the host gunzip program. 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 in 1 second of CPU time when using the Rice compression algorithm.

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. See our PASP paper for more details.

6. Compression of floating point FITS images

It is generally not practical to losslessly compress FITS images in floating point format (with BITPIX = -32 or -64). This is because most of the compression algorithms do not support floating point data, and even if they do (e.g., gzip), a large fraction of the bits in the mantissa of the image pixel values are often filled with uncompressible noise, which severely reduces the file compression ratios. For this reason, fpack always converts the pixel values into 32-bit integers using a linear scaling function:

integer_value = (floating_point_value - ZERO_POINT) / SCALE_FACTOR

This array of scaled integers is then compressed using the specified compression algorithm (usually Rice). 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 SCALE_FACTOR in the scaling function controls how closely the inverse scaled values approximate the original floating point values: decreasing SCALE_FACTOR reduces the spacing between the quantized levels in the inverse-scaled values and thus more closely reproduces the original pixel values. However, this also magnifies the dynamic 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 more precision or achieving greater compression.

It is not easy to directly determine an appropriate SCALE_FACTOR value to use with a given image, therefore fpack provides instead a quantization parameter called "q" for specifying how closely the inverse-scaled integer pixel values must approximate the original floating point pixel values, relative to the measured noise in background areas in the image. The image pixel values will be quantized so that the spacing between the adjacent discrete levels is equal to the measured R.M.S. noise in the background regions of the image divided by q. In other words, the pixel values are recorded with a factor of q times more accuracy than the measured background noise level. Formally, the number of noise bits that are preserved in each pixel value is given by $\log_2(q) + 1.792$. Thus, when using the default value of q = 16 (which preserves about 5.8 binary bits of noise in each pixel value), if the RMS noise in a tile of floating point image has a value of 25.0, then the pixel values in the compressed image will be quantized into levels that are separated by intervals of 25 / 16 = 1.56. The maximum difference between the pixel values in the compressed image and the original image will be half this value. Increasing the value of q will produce compressed images that more closely approximate the pixel values in the original floating point image, but will also increase the size of the compressed image file, as shown in Table 2 which gives the approximate compression ratio that can be expected for a given q value. Further details of this floating point compression scheme are given in an ADASS paper by White and Greenfield, 1999.

In some cases it may be preferable to specify the actual quantization factor to be used when converting the floating point pixels to scaled integers, rather than giving the q value relative to the calculated noise in each tile. Doing this will ensure that every tile will use exactly the same quantization factor, and it will also speed up the compression process because it is not necessary to calculate the noise level in each tile. This can be done by specifying the negative of the absolute quantization factor for the value of q. In this case, a smaller absolute value (e.g. -.001 instead of -.01) will produce compressed images that more closely approximate the pixel values in the original floating point image, but will also increase the size of the compressed image file. Note that this option of specifying a negative q value is not recommended for general use,; it should only be used in cases where the noise properties in the input image are well understood.

Table 2. Floating Point Images

q	**Compression Ratio**
4	6.0
8	5.1
16	4.4
32	3.9
64	3.5

Once the floating-point pixel values have been converted to scaled integers, they may be compressed using either the Rice, GZIP, or H-compress algorithms. In most cases the default Rice algorithm provides the best compromise between speed and compression factor.

It must be emphasized that each fpack user is ultimately responsible for determining the appropriate q value to use when compressing floating point images; using too small a value of q (or a negative q value that is too large) could result in unrecoverable loss of some of the information that was present in the original floating point image. This loss may be acceptable in cases where the image is only used for qualitative purposes. Using too large a value of q, on the other hand, will simply preserve more of the noise in the image and reduce the amount of compression that is achieved. Anecdotally, tests performed at the Space Telescope Institute and elsewhere using q = 16 did not detect any significant difference between various photometric or astrometric quantities in a sample of compressed astronomical images, as compared with the same quantities derived directly from the original uncompressed image. This is no guarantee, however, that the default value of q = 16 is suitable for all applications, so users are strongly urged to perform quantitative tests using different values of q to determine the appropriate level to use for their particular application.

7. fpack command-line parameters

The fpack program is invoked on the command line like other host-level utility programs:

```
fpack [OPTION]... [FILE]...
```

The specified options must appear before the list of files to be compressed. The file names may contain the usual wildcard characters that will be expanded by the Unix shell.

The compression algorithm to use is selected with one of the following options:

-r Rice [default], or
-h Hcompress, or
-g GZIP (per-tile), or
-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} (16777216).
-d no compression (debugging mode)

Tiling pattern specification:

When using the Rice, GZIP, or PLIO compression algorithms, the default tiles each contains 1 row of the image (i.e., the tiles are one dimension and contain NAXIS1 pixels). The Hcompress algorithm requires that the tiles be 2-dimensional, therefore the default is to use 16 rows of the image per tile. If this would cause the last partially full 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 more equal 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
-t <axes> comma separated list of tile dimensions (e. g., -t 200,200 will produce tiles

that are 200 x 200 pixels in size)

Compression parameters for floating point images:

-q <level> Quantization level when compressing floating point images. Default value = 16. See the previous section on compressing floating point images for more discussion of this parameter. It is important to realize that fpack does not exactly preserving the original pixel values when compressing floating point images. Users should carefully evaluate the compressed images (e.g., by uncompressing them with funpack) to make sure that any essential information in the original image has not been lost.

Positive q values are interpreted as relative to the R.M.S. noise in the image. A larger q value will more closely preserve the original pixel values, and will result in less compression. 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, and all the images in a dataset, are compressed using the identical value, regardless of slight variations in the measured noise level. This can be done by specifying the negative of the desired value. In this case, smaller absolute values of q (e.g., -.001 instead of -.01), will better preserve the original pixel values, at the expense of smaller (worse) compression ratios. The -T option (described below) can be used to calculate the noise level in the image.

-n <noise> This rarely used parameter rescales the pixel values in a previously scaled image to improve the compression ratio by reducing the R.M.S. noise in the image. This option is intended for use with 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 (e.g., -32768 to +32767 for 16-bit integers and -2147483648 to +2147483647 for 32-bit integers). The measured R.M.S. noise in these integer images is typically so huge that they cannot be effectively compressed. This -n option rescales the pixel values so that the R.M.S. noise will be equal to the specified value. Appropriate values of n will likely be in the range from 8 (for low precision and the high compression) to 64 (for the high precision and lower compression). Users should read the section on compressing floating point images, above, for guidelines on choosing an appropriate value for n that does not lose significant information in the image.

Parameters for lossy compression of integer images:

-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 R.M.S. 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 dataset, are compressed with the identical scale value, regardless of slight variations in the measured noise level. This is done by specifying the negative of the desired value (e.g. -30., which would be equivalent to specifying a scale value of 2.0 in an image that has RMS noise = 15.).

It is important to realize that this option achieves the high compression ratios at the expense of not exactly preserving the original pixel values in the image. Users should carefully evaluate the compressed images (e.g., by uncompressing them with funpack) to make sure that any essential information in the

image has not been lost.

The compressed output file name is usually constructed by appending ".fz" to the input file name, and the input file is not deleted, but this behavior may be modified with the following parameters:

-F force the input file to be overwritten by the compressed file with the same name. This is only allowed when a lossless compression algorithm is used.

-D delete the input file after creating the compressed output file.

-Y suppress the prompts to confirm the -F or -D options

-S output the compressed FITS file to the STDOUT stream (to be piped to another task)

Other miscellaneous parameters:

-v verbose mode; list each file as it is processed

-R <filename> write the comparison test report (produced by the -T option) to a file in a format that is suitable for further analysis.

-L list all the extensions in all the input, files. No compression is performed.

-C don't update FITS checksum keywords

-H display a summary help file that describes the available fpack options

-V display the program version number

-T produce a report that compares the compression ratio and the compression and uncompression times for each of the main compression algorithms. The input file remains unchanged and is not compressed. The report is similar to the following:

```
File: ct655046_13.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 first line of the report gives the name of the FITS file; the 3rd line gives the following parameters:

Ext – extension number within the file (zero based)

BITPIX – FITS datatype 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 = $\log_2$(noise3) + 1.792

MaxR – theoretical maximum possible compression ratio = BITPIX / Nbits

This is followed by a table with 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 (i.e., does it exactly preserve the pixel values)?

The following 4 parameters give the measured image read rates, in units of seconds/MB

ElpN – elapsed time to read the entire image with a single subroutine call

CPUN – CPU time to read the entire image with a single subroutine call

Elp1 – elapsed time to read the whole image, one row at a time

CPU1 – CPU time to read the whole image, one row at a time

The rows in this table correspond to the following cases:

Native – this just gives the read speed of the input uncompressed image

Rice – when using the Rice compression algorithm

Hcomp – when using the Hcompress algorithm

GZIP – when using the gzip algorithm (within the FITS tiled image compression format)

None – the image is simply tiled and packed into the FITS tiled image format, without performing any compression on the tiles.

7. funpack command-line parameters

funpack shares many of the same parameters as fpack as shown below:

Output file naming parameters:

-F force the input file to be overwritten by the uncompressed file with the same name. This is only allowed when a lossless compression algorithm is used.

-D delete the input file after creating the compressed output file.

-P <pre> prepend the <pre> string to the input file name to generate the name of the uncompressed output file.

-O <name> used to specify the full name of the uncompressed output file.

-S output the uncompressed FITS file to the STDOUT stream (to be piped to another task)

-Z recompress the output file with the host gzip program

Other miscellaneous parameters:

-v verbose mode; list each file as it is processed

-L list all the extensions in all the input, files. No uncompression is performed.

-C don't update FITS checksum keywords

-H display a summary help file that describes the available funpack options

-V display the program version number

8. Building fpack and funpack

The latest versions of the fpack and funpack C source code are available from http://heasarc.gsfc.nasa.gov/fitsio/fpack. These programs are also included in the CFITSIO source file distributions (but not necessarily the latest version) available at http://heasarc.gsfc.nasa.gov/fitsio.

To build the software on unix systems, first download and build the CFITSIO library. If necessary, untar the latest version of the fpack and funpack source code into the CFITSIO directory, overwriting the older version. Then enter the commands

```
make fpack
make funpack
```

in that directory. This will create the fpack and funpack executable files which may be copied to any other suitable directory (e.g. the local /bin directory).