Fpack and Funpack Utilities for FITS Image Compression and Uncompression

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

20 August 2008

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 (as long as 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 algorithms.

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 and provided general advice on other image compression issues.

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, using the Rice algorithm, generally provides higher compression ratios and faster

compression and uncompression 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 even greater, however, when the user's analysis software is capable of directly reading and writing the files in the compressed form. For example, the widely used ds9 image display program supports the tile compressed format. In this scenario, the file does not need to be physically uncompressed. 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 slightly more CPU resources, but this is at least partially offset by a gain in I/O performance because fewer bytes of data need to be transfered to or from disk storage.

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 partial 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 for compatibility with 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 random noise that is present in the pixel values in the image. As the amount of noise is reduced, the compression ratio increases exponentially. By comparison, other image attributes, such as the mean pixel value, or the presence of image structures such as stars, spectral lines, or large scale gradients across the image, have little or no effect on the compression factor.

In order to quantify this effect, fpack uses a robust algorithm for measuring the amount of noise in "background" areas of the image. This algorithm was originally developed to measure the signal-to-noise in spectroscopic data (see the June 2007 ST-ECF Newsletter). In particular, we use the 3rd order "median absolute difference" formula to compute the standard deviation of the (assumed) Gaussian distributed noise for the pixels in each row of the image:

$$\text{standard deviation} = 0.6052 * \text{median}(\text{abs}(-x(i-2) + 2x(i) - x(i+2)))$$

where x(i-2) is the value of the pixel 2 spaces to the left of pixel i, and x(i+2) is the value of the pixel 2 spaces to the right, and the median value is computed over all the pixels in each row of the image. This formula is much less sensitive to outlying pixel values than the usual formula for computing the standard deviation of a sample of values. The overall noise value for the image as a whole is then computed from the mean of the median values for each row of the image. The -T test option in fpack can be used to compute the random noise level in any image.

Extensive testing on synthetic images with known amounts of Gaussian noise, as well as on large sets of real astronomical images, shows that the amount of compression that is achieved can be accurately predicted from the noise value that is computed with the above formula. Figure 1 shows that the compression ratio (using the Rice algorithm in this case) is very tightly correlated with the measured noise level. The points represent a complete set of 1632 16-bit integer CCD images that were obtained with a mosaic camera (containing 16 individual CCD sensors) during one night of observing at KPNO. Each CCD image contains 1112 by 4096 16-bit integer pixels totaling 9.1 MB in size. Most of scatter seen in this figure is caused by slight differences in the imaging characteristics of the 16 different CCD detectors in this mosaic camera.

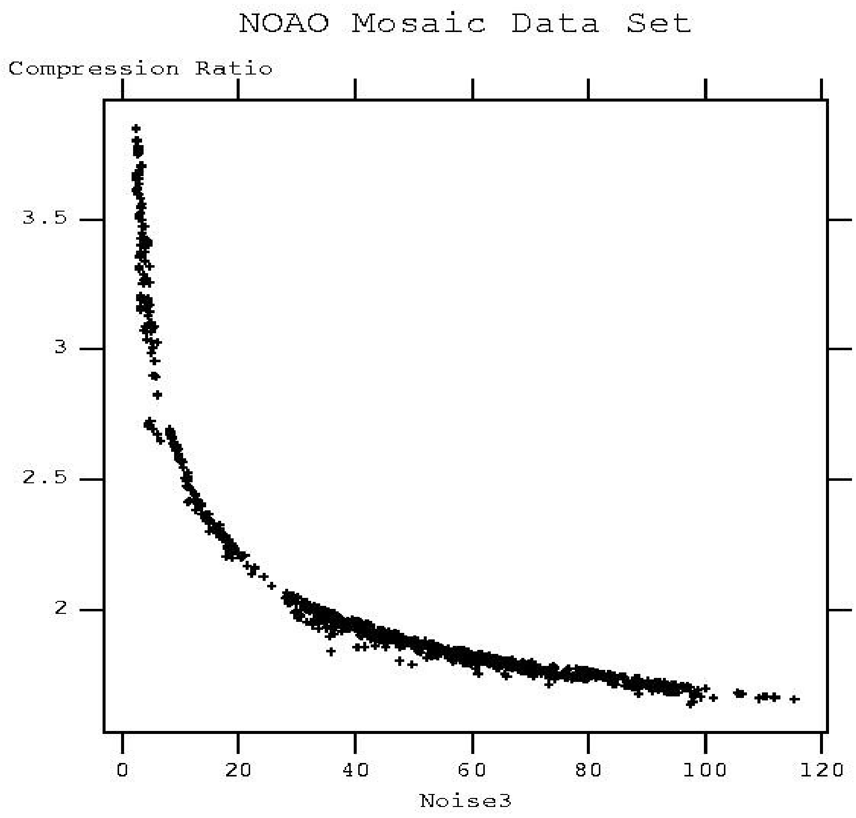

Figure 1. Rice compression ratio versus the measured noise in a set of 16-bit CCD images obtain during one night of observing with the NOAO mosaic camera.

In the typical photon counting type detectors used in astronomy, the noise in each pixel value scales with the square root of the number detected photons. Thus, the amount of noise in the image naturally increases with the mean value of the pixels. The practical implication of this fact is that the different types of exposures taken during a typical astronomical observing session (e.g., bias frames with low pixel count values, flat field images with large pixel values, and images of the sky taken with varying exposure times) will have distinctly different amounts of noise and hence will compress by differing amounts. In Figure 1, the points with the smallest amount of noise and a compression factor greater than 2.7 correspond to the "bias" CCD images that have zero exposure time and very low count levels. The intermediate set of points, with compression ratios in the range 2.1 to 2.7 and noise values in the range 5 to about 28, correspond to short exposures of 10 to 30 seconds in duration taken mainly to optimize the telescope focus. The remaining points, with noise values greater than 28 and compression ratios slightly less than 2.0 correspond to the longer exposure image of the night sky and the flat field exposures of the interior of the illuminated dome which also have high count levels. It is an unfortunate fact of nature that the more scientifically interesting images (e.g., the deep exposures of the sky) are generally noisier and will not compress nearly as well as the less interesting bias and focus calibration images.

5. Comparison of integer lossless compression algorithms

The data set of 1632 16-bit integer CCD images shown in Figure 1 was used to compare the speed and file compression ratios for the 3 different compression algorithm that are currently supported by fpack, namely, Rice, GZIP, and Hcompress. For comparison, we also measured speed and compression factor when using the host-level GZIP program to externally compress the entire FITS file. When using GZIP we always use the "-1" fastest compression option, which can nearly double file compression speed without significantly increasing the size of the compressed file. One small disadvantage of using this option is that it increases the file uncompression time by a few percent.

The mean file compression ratios and the relative compression and uncompression elapsed CPU times for the 4 different compression methods are shown in Table 1. These figures are the mean for all 1632 images in the sample data set, and the CPU times in each case are relative to those for the Rice algorithm. As an aside, all the time measurements discussed here refer to the CPU times because the total elapsed times, which in principle include the disk read and write time as well as the CPU times, proved to be difficult to measure with any consistency. In practice, the elapsed times were usually nearly equal to the CPU times.

Table 1. 16-bit integer image compression

	Rice	**Hcompress**	**GZIP**	**Host GZIP**	**Host GZIP -1**
Compression Ratio	2.11	2.18	1.53	1.64	1.64
Relative compression CPU time	1.0	2.8	5.6	4.9	2.8
Relative uncompression CPU time	1.0	3.1	1.9	0.9	0.9

The first row in Table 1 shows that 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. Compared to Rice, Hcompress produces compressed files that are about 3% smaller, however, for most applications this slight gain is not worth the much longer CPU times required to compress and uncompress the images. Hcompress is mainly useful for performing lossy compression of the image, where much higher compress ratios can be achieved by not exactly preserving the pixel values.

Image compression using the Rice algorithm is 3 to 6 times 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 implementation of the GZIP algorithm in CFITSIO requires that the input and output files be copied to and from intermediate storage buffers in memory.

The uncompression benchmarks shown in the third row of Table 1 are arguably more important that the compression numbers because in many instances an image only has to be compressed once (by the data provider) but it must be uncompressed by every user, sometimes multiple times if the analysis software can directly read the image in it's compressed form. Here again, the Rice algorithm is 2 to 3 time faster than the other 2 methods offered by fpack/funpack. While the host-level GUNZIP program does provide about the same file uncompression speed as the Rice algorithm in fpack, the fact that the GZIP compressed files are larger, and host-level GZIP does not provide all the other advantages of using the tiled image compression scheme discussed in the previous Section 2, makes Rice the much better choice for most applications.

The actual elapsed compression time for a given image depends on the speed of the underlying computing system, but as a point of reference, on a 2006 vintage Linux machine with a 2.4 GHz AMD Opteron 250 dual core processor, fpack can compress or uncompress a 50 MB 16-bit integer image with the Rice algorithm in 1 second of CPU time.

The above performance tests were done with 16-bit integer images, but very similar rules apply when compressing 32-bit integer images. The main difference is that 32-bit images compress twice as well as 16-bit images, assuming the same amount of random noise in both images. In other words, if a 16-bit image and a 32-bit image have the same noise level, then the 2 compressed images will have almost exactly the same size. Since the 32-bit image was originally twice as big as the 16-bit image (given the same image dimensions), the compression ratio for the 32-bit image will be twice that of the 16-bit image. In practice, however, the noise level in 32-bit images will tend to be greater than in 16-bit images (simply because the mean pixel level is higher) which will hurt the compression ratio.

Table 2 compares the performance of the different compression methods on a set of 1788 32-bit integer images taken during on night with the NOAO NEWFIRM camera. The mean noise level of 67.7 in these images is larger than in the 50.1 value in the previous 16-bit integer image data set, hence the mean compression ratio for these images is a little less than twice that of the corresponding value for the 16-bit images.

Table 2. 32-bit integer image compression

	Rice	Hcompress	GZIP	Host GZIP	Host GZIP -1
Compression Ratio	3.76	3.83	2.30	2.50	2.32
Relative compression CPU time	1.0	5.2	7		4.7
Relative uncompression CPU time	1.0	3.4	2.2	1.0	1.3

6. Specific fpack compression guidelines

The optimal compression technique to use with fpack depends on the data type of the FITS image (either integer for floating point) and on whether lossless or lossy compression is to be performed. The following subsections provide guidelines on using fpack in each of these cases.

6.1 Lossless compression of integer FITS images

As was shown in section 5, the default Rice compression algorithm usually provides the optimum trade off between compression ratio and speed when losslessly compressing integer images (with BITPIX = 8, 16, or 32,).   The image tiling pattern does not have much affect on the compression, so in most cases the default row-by-row tiling pattern that fpack uses with the Rice algorithm should be used. Users can experiment using different compression algorithms (-g for GZIP, -h for Hcompress, and -p for the IRAF PLIO algorithm) or different tiling patterns (with the -t option) to test how these affect the compression ratio and speed. The fpack program also provides the -T test option that is convenient for comparing the compression ratio and the amount of time needed to compress and uncompress the image when using each of the 3 main compression algorithms (Rice, Hcompress, and GZIP).

6.2 Lossy compression of integer images

The H-compress algorithm may be used in a "lossy" mode to achieve higher compression of integer images. This mode will obviously not exactly preserve the pixel values in the original image, but this may be appropriate if the image is very noisy, or if the image is intended mainly for qualitative purposes. To use this option in fpack, choose the H-compress algorithm, and set the -s option to a positive value, typically somewhere in the range 1.0 to 100. Larger values will produce greater amounts of compression. Note that it may be necessary to increase the default tile size when doing lossy compression, otherwise the boundaries of the tiles may become visible in the compressed image.

6.3 Lossy compression of floating point FITS images

When compressing floating point FITS images which have BITPIX = -32 or -64, 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. 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 individually for each tile of the image. This scaling of floating point images is performed because a) the Rice and Hcompress algorithms can only operate on integer arrays, and b) even if the algorithm can compress floating point arrays (e. g., GZIP), the amount of compression that is achieved is generally poor as compared to compressing the equivalent array of scaled integers.

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, as discussed previously, adversely affects the amount of compression that is achieved. Thus, there is a direct trade-off between providing more precision (by decreasing SCALE_FACTOR) or achieving a greater amount of compression (by increasing SCALE_FACTOR).

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. With the default value of q = 16, 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 16. In other words, the pixel values are recorded with a factor of 16 times more accuracy than the measured background noise sigma. For example, 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 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. Further details of this floating point compression scheme are given in the ADASS paper by White and Greenfield, 1999.

Once the floating-point pixel values have been converted to scaled integers, they are 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. Rice is generally faster and provides better compression than GZIP. While H-compress can often produce slightly better compression ratios than the Rice algorithm, this small benefit is usually not worth the much larger CPU times that are required to compress or uncompress the image using H-compress.

The q parameter directly determines the amount of compression in floating point images because the noise in the scaled integer array is numerically equal to q (so for example, if q = 16 then the noise in the scaled integer image will also equal 16). Table 2 shows the approximate compression ratio that will be achieved for floating point images as a function of q when using the default Rice compression algorithm. As was noted in section 4. these compression ratios for the scaled 32-bit integer images are approximately twice those shown in Table 1 for 16-bit integer images with the equivalent noise level.

Table 2.

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

It must be emphasized that each fpack user is responsible for determining the appropriate q value to use when compressing floating point images; using too small a value of q 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 random 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 default behavior of fpack is to create an output compressed file with the same name as the input file, but with a ".fz" suffix (e.g., "image.fits" becomes "image.fit.fz"). The originally image file is not deleted, unless the -D flag is specified. Similarly, funpack expects the input compressed image to have a ".fz" extension, and creates an output uncompressed filename without the extension. Alternatively, the "-F" flag may be specified to force the input file to be overwritten by the output file with the same name. If the image is compressed with a lossy algorithm, then the user will be prompted to confirm whether to delete the input file. These confirmation prompts can be suppressed by specifying the "-Y" flag.

Compression algorithm: select 1 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). Note that the PLIO algorithm cannot be used with FITS images that have unsigned integer values because the pixel values in these images are internally shifted into the range of negative signed integers.

-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.

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. See the section 5.3 on compressing floating point images for a description 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 derived using the formula described in section 4. 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. The -T option can be used to calculate the noise level in the image.

-n <noise> Rescale 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 nearly 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 (because the compression ratio depends directly on the noise). 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.

Output file naming 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 deleting the input file when using -F or -D

-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
-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
-R <filename> write the comparison test report (produced by the -T option) to a file in a format that is suitable for ingest into a data base 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

8. 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 uncompressed output file.
-Y suppress the prompts to confirm deleting the input file when using -F or -D
-P <pre> prepend the <pre> string to the input file name to generate the name of the uncompressed output file.
-O <name> specifies the full name of the output uncompressed file.
-S output the uncompressed FITS file to the STDOUT stream (to be piped to another task)
-Z recompress the output file with gzip

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

9. 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).