

The KB1 Image Compression System

Andy Fraser

September 7, 2001

Abstract

KB1 (Kompress Bits: version 1) was conceived of as a demonstration and template of a structure for image compressors which would permit effort to be focused on modeling. Although KB1 itself uses the first and easiest model that occurred to me, its performance is within 10% of JPEG2000 on the test images. This document describes how I implemented KB1 and how well it performs on the test images.

1 Implementation of Compression

My goal in writing KB1 was to isolate the modeling effort and code so that future refinements can focus on modeling exclusively. Figure 1 shows how data moves in KB1. Ignoring Erik Bollt's stated aversion to multi-lingual projects, I've used four languages in KB1:

Python: I used python for the analysis block because it does string manipulation, makes system calls, and calls other programs.

C: The arithmetic encoder is implemented in C because there was modular source code on the net.

Octave: In the analysis block, I used an octave script called *fit.m* to fit model parameters. I wrote another script, *model1.m* to produce CDFs for the arithmetic encoder (or decoder). Plugging variants of these two scripts into the KB structure will be easy *no matter what language they are implemented in.*

encode1.sh

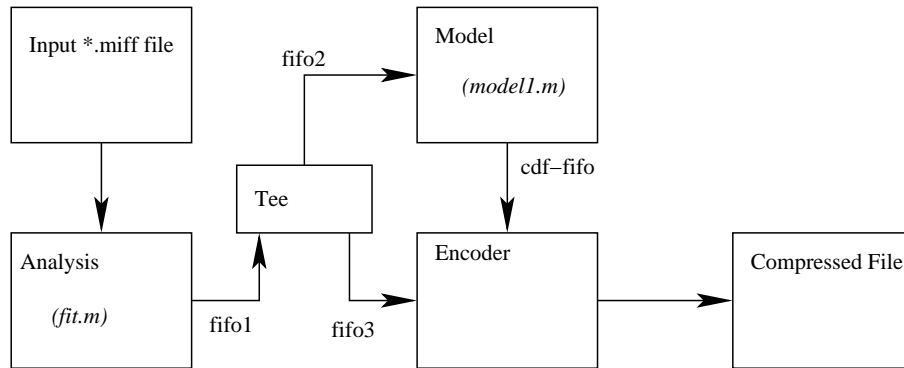


Figure 1: Data flow diagram for the KB1 compressor. The flow is coordinated by the script *encode1.sh*. The blocks are implemented as follows: Analysis; *analysis1.py* and *fit.m*. Model; *model1.m*. Encoder; *fwac_en* (a compiled C program). One can test new modeling ideas by modifying the files *fit.m* and *model1.m* without changing any other programs or scripts.

Shell: The script *encode1.sh* uses FIFOs and a tee to set up the data paths. This is the entire script:

```
# encode1.sh
# Syntax: sh encode1.sh miff_file compressed_file

INFILE=$1
OUTFILE=$2

rm *fifo*
mkfifo fifo1
mkfifo fifo2
mkfifo fifo3
mkfifo cdf-fifo
tee fifo2 >fifo3 <fifo1 &

python analysis1.py $INFILE fifo1 &
octave model1.m &
fwac_en fifo3 cdf-fifo $OUTFILE
```

1.1 Example and Formats

Here is an invocation of the KB1 compressor:

```
>sh encode1.sh 01.miff 01.amf
```

The input file, *01.miff*, is in the ImageMagick uncompressed format and the output file *01.amf* is in the compressed KB1 format. I chose the ImageMagick uncompressed format because it is easy to parse. Data is passed between the components shown in Fig. 1 as follows:

Byte Streams: The data passing through *fifo1*, *fifo2*, and *fifo3* are ASCII decimal digit representations of bytes. There is only one number on each line.

CDF Stream: The data passing through *cdf-fifo* is a sequence of cumulative distribution functions. Each function is given on a single line which consists of 257 ASCII decimal integers. The first number, N_0 is always 0, the other numbers must increase monotonically: $N_k \leq N_j$ if $k < j$, and $N_{256} \leq 16383$. In KB1, $N_{256} = 16383$. The probability of byte value b is $P(b) = \frac{N_{b+1} - N_b}{N_{256}}$.

Named Files: The analysis block writes model parameters and the image header to files in a directory named *tmp/parameters*. Contrary to the intuition of any Bavarian skeptics, I am not using these files to cheat. Information sufficient to construct these files passes through *fifo2* to the model block before the model uses any information in the files. In the image recovery procedure, these files are re-created from information in the compressed data.

1.2 Analysis

The analysis block fits a parametric model (see Fig. 2) to the entire image and constructs a data stream or sequence of bytes to be compressed. I designed the sequence so that as it is decompressed, the information required to model each byte will be available in its predecessors. Figure 3 outlines the procedure.

x_1	x_2
x_3	y

$$\begin{aligned}\hat{y}_r &= a_{r,1} \cdot x_{r,1} + a_{r,2} \cdot x_{r,2} + a_{r,3} \cdot x_{r,3} + \\ &\quad a_{r,4} \cdot x_{g,1} + a_{r,5} \cdot x_{g,2} + a_{r,6} \cdot x_{g,3} + \\ &\quad a_{r,7} \cdot x_{b,1} + a_{r,8} \cdot x_{b,2} + a_{r,9} \cdot x_{b,3}\end{aligned}$$

$$\begin{aligned}\hat{y}_g &= a_{g,1} \cdot x_{r,1} + a_{g,2} \cdot x_{r,2} + a_{g,3} \cdot x_{r,3} + \\ &\quad a_{g,4} \cdot x_{g,1} + a_{g,5} \cdot x_{g,2} + a_{g,6} \cdot x_{g,3} + \\ &\quad a_{g,7} \cdot x_{b,1} + a_{g,8} \cdot x_{b,2} + a_{g,9} \cdot x_{b,3} + \\ &\quad a_{g,10} \cdot y_r\end{aligned}$$

$$\begin{aligned}\hat{y}_b &= a_{b,1} \cdot x_{r,1} + a_{b,2} \cdot x_{r,2} + a_{b,3} \cdot x_{r,3} + \\ &\quad a_{b,4} \cdot x_{g,1} + a_{b,5} \cdot x_{g,2} + a_{b,6} \cdot x_{g,3} + \\ &\quad a_{b,7} \cdot x_{b,1} + a_{b,8} \cdot x_{b,2} + a_{b,9} \cdot x_{b,3} + \\ &\quad a_{b,10} \cdot y_r + a_{b,11} \cdot y_g\end{aligned}$$

Figure 2: Pixel forecasts. \mathbf{a}_c denotes the vector of autoregressive coefficients for constructing \hat{y}_c , the central forecast value for color c . The equations for these central forecast values appear below the matrix/grid that describes the subscripting/naming convention for the relative pixel locations.

analysis1.py

Read the image file.

Separate out the header and body. Save in *tmp/parameters/header* and *tmp/body.txt*.

Call octave script **fit.m**:

Read *tmp/body.txt* and write the following into *tmp/parameters/data*:

n_x, n_y The number of rows and columns in the image.

$R_{\text{left}}, R_{\text{right}}, G_{\text{left}}, G_{\text{right}}, B_{\text{left}}, B_{\text{right}}$ The top row and left column of pixel values.

a_r, a_g, a_b The coefficients of the linear forecasts of the red, green, and blue pixel values respectively.

N_r, N_g, N_b Histograms of the respective forecast errors.

Create compressed file *par.tar.bz2* that contains both the image header and the octave data.

Write the following sequence of bytes to the output stream:

4 bytes giving the length of *par.tar.bz2*.

All of the bytes in *par.tar.bz2*.

All of the bytes in the body of the image.

Figure 3: Analysis procedures implemented in *analysis1.py* and *fit.m*. I use *gzip* to compress everything except the body of the image, ie, the header, the top row, the left column, and the model parameters.

1.3 Model

I implemented the model required by the arithmetic coder and decoder in the octave script *modell.m*. The only details of its operation that may require more explanation than the overview in Fig. 4 concern the CDFs for the pixel values in the body of the image.

modell.m

```
Do 4 times to get tar-file length:
    Write a uniform CDF.
    Read a byte.
For each byte of the tar-file:
    Write a uniform CDF.
    Read a byte.
Read the parameter file.
Sum histograms to get CDFs.
For each pixel location in raster order:
    For each color:
        Write a forecast CDF.
        Read a byte.
```

Figure 4: Steps implemented by the script *modell.m*. The model forecasts uniform distributions for the bytes in the compressed tar-file (thus costing 8 bits per byte). It will forecast the exact values for the top row and left column (thus costing zero bits per byte). It will use the model parameters to incrementally forecast each byte in the remainder of the image body.

For those byte values that are transmitted ahead of the image body as part of the header and parameter information, the model produces step function CDFs. To describe how I produce a CDF for each of the remaining bytes in the body, let t denote the byte position (l) and the color (c). Let y_t be the byte value, \mathbf{x}_t the context to be used, \mathbf{a}_c the relevant vector of autoregressive

coefficients, and

$$\hat{y}_t = \lceil \mathbf{x}_t \cdot \mathbf{a}_c - \frac{1}{2} \rceil \text{ (result rounded to integer)}$$

the central forecast value. (See Fig. 2 for some clarification of the location in the image of the context values \mathbf{x}_t .) For each color c , I can use a single histogram of the “errors”, $\epsilon_l = y_l - \hat{y}_l$, that occur over the entire image. The histogram, call it $\{C_c(\epsilon) : \epsilon \in [-256, 255]\}$, is available as part of the parameters that *fit.m* has collected. For each $t = (l, c)$, I send a CDF that corresponds to C_c shifted by \hat{y}_t and clipped to require that $y \in [0, 255]$.

Worth
checking
that his-
tograms

1.4 Arithmetic Coder and Decoder

The arithmetic coder and decoder are sketched in Fig. 5 and Fig. 6 respectively. The guts of both are available at <http://www.cipr.rpi.edu/~wheeler/ac>. I only wrote wrapper programs that handle ASCII/Decimal data and flush the I/O buffers as necessary.

are
not re-
flected.

fwac_en.c

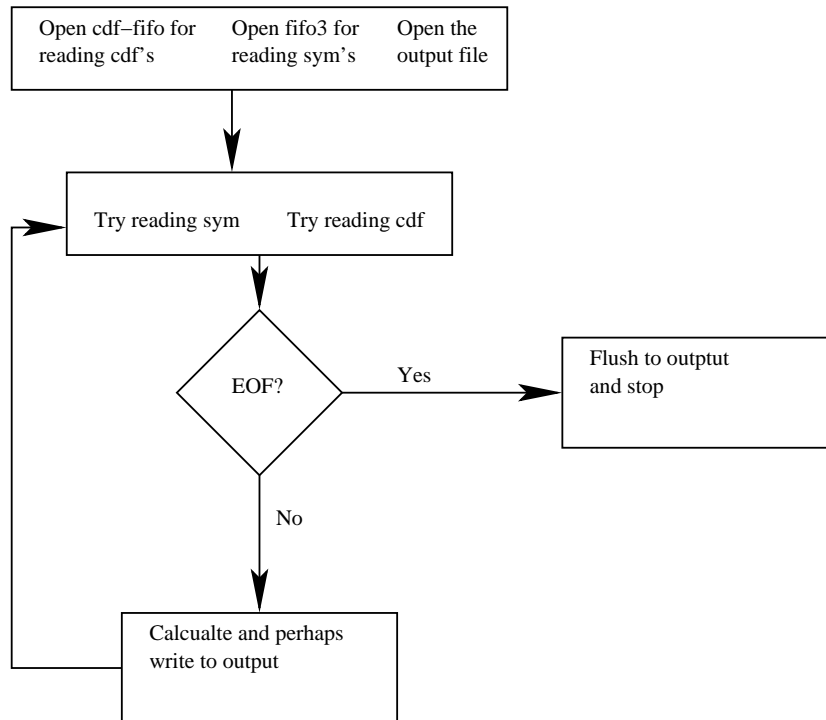


Figure 5: Flow chart for the arithmetic encoder used in the KB1 compressor. It is implemented in the program *fwac_en* (Fred Wheeler Arithmetic Encoder).

fwac_de.c

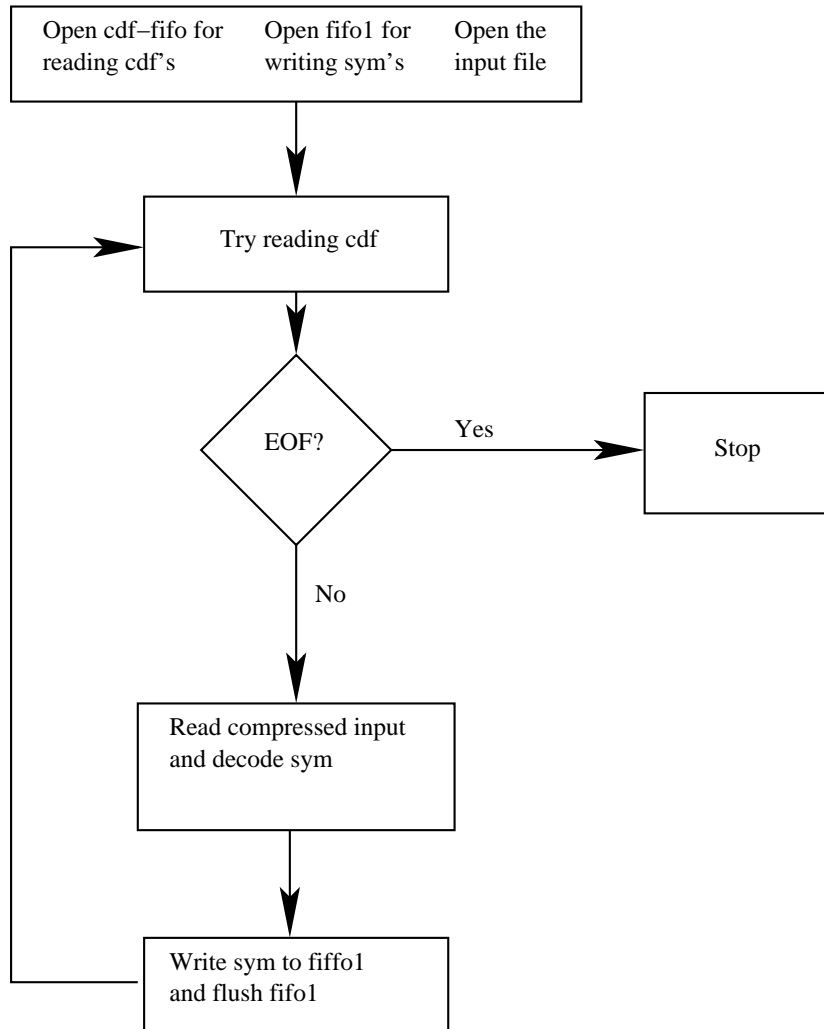


Figure 6: Flow chart for the arithmetic decoder used in the KB1 decompressor. It is implemented in the program *fwac_de* (Fred Wheeler Arithmetic Decoder).

2 Implementation of De-Compression

I designed the compression procedure to be reversible. Since the data streams that the model reads and writes are exactly the same for compression and de-compression, I can use the same octave script, *model1.m*, in both contexts. The handshaking/synchronization is more critical for de-compression than compression. In particular, the synthesis procedure must make the parameter files available to the model before the image body can be decompressed. But overall, if one understands the compression procedure, the de-compression procedure will seem simple.

Data flow for decompression is sketched in Fig. 7, and Fig. 8 outlines the synthesis script

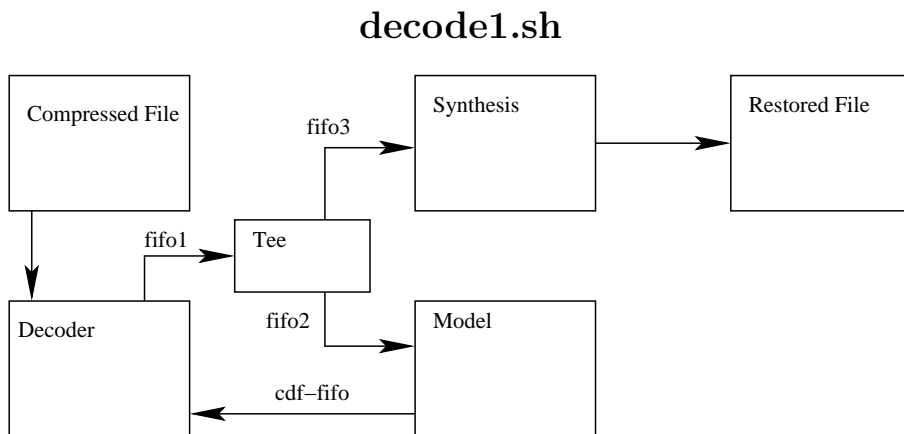


Figure 7: Data flow diagram for the KB1 decompressor. The flow is coordinated by the script *decode1.sh*. The blocks are implemented as follows: Decoder; *fwac_de*. Model: *model1.m*. Synthesis: *synthesis1.py*.

synthesis1.py

1. Read the length of the tar-file.
2. Read the tar-file.
3. Extract parameters from the tar-file so that they are in known locations. (`modell.m` must have access to some of these immediately.)
4. Read body of the image.
5. Write the header and the body of the image to the recovered image file.

Figure 8: Synthesis procedures implemented in `synthesis1.py`. In the figure, *Read* means read from input symbol stream. (The script `decode1.sh` uses `fifo3` for that stream.) The synchronization of step 3 is critical because the model, `modell.m`, must use the extracted parameters to produce the CDFs that the arithmetic decoder, `fwac_de`, uses to decompress the body of the image.

3 Compression Performance

Image #	KB1 Size	JPEG2000 Size	JPEG/KB1
07	471347	418078	0.89
23	468817	418118	0.89
14	559402	499507	0.89
24	555890	499838	0.90
15	490652	442301	0.90
03	438689	397818	0.91
11	503099	456809	0.91
22	545972	496233	0.91
05	581680	531759	0.91
21	519592	479921	0.92
04	493285	460142	0.93
19	517538	482853	0.93
06	505252	471519	0.93
09	476089	445081	0.93
12	453803	425637	0.94
18	582321	546908	0.94
02	478161	450465	0.94
17	477648	451930	0.95
01	539402	510509	0.95
20	418606	397094	0.95
10	475935	453194	0.95
08	574015	547596	0.95
16	451579	431393	0.96
13	606258	583077	0.96

Table 1: Comparison of performance of KB1 to JPEG2000 on 24 test images. The sizes are in bytes. For image 01, the bziped tar-file is 1754 bytes. So parameter overhead is not significant.