Compressing Elevation Data[*]

To appear, Symposium on Spatial Data, Orono, Maine, 1995

Wm Randolph Franklin

Electrical, Computer, and Systems Engineering Dept., 6026 JEC,
Rensselaer Polytechnic Institute, Troy, New York 12180–3590
(518) 276–6077, Fax: (518) 276–6261
wrf@ecse.rpi.edu, http://www.ecse.rpi.edu/homepages/wrf/

May 20, 1995

Contents

[*]Partial support was provided by NSF grant CCR-9102553.

List of Figures

Abstract. This paper compares several, text and image, lossless and lossy, compression techniques for regular gridded elevation data, such as DEMs. `Sp_compress` and `progcode`, the best lossless methods average 2.0 bits per point on USGS DEMs, about half the size of `gzip`ped files, and 6.2 bits per point on ETOPO5 samples. Lossy compression produces even smaller files at moderate error rates. Finally, some technques for compressing TINs will be introduced.

1 Introduction

Since Digital Elevation Model (DEM), or, similarly, Digital Terrain Elevation Data (DTED), files are so large, minimizing storage space is important. Altho storage space becomes ever cheaper, the data has higher and higher resolution. Besides the cost of the space, the data file size also affects transmission time and cost. Even within one computer, since processors are getting faster faster than busses and memories are getting faster, smaller data can lead to faster computation. Therefore, altho the question has been studied by many people since computers were first used in cartography, it is still appropriate to ask: How should we compress elevation data?

The first decision is whether to use a general- or a special-purpose compression algorithm; this is not at all clear. The special-purpose method, such as a Triangulated Irregular Network (TIN), can take advantage of the peculiar nature of elevation data, and of the desired error behavior. However, a general-purpose algorithm might have so much effort devoted to its development that it might perform quite well on elevation data, even though not designed specifically for that. In fact, in computer engineering, special purpose machines have generally been failures for that reason. Such failures include most of the machines developed in all the following categories: Lisp machines, floating point processors, database engines, special graphics engines, and parallel machines.

In this paper, we test compress a regular gridded sample digital elevation file with some older, generally available, compression algorithms, such as `compress`, `gzip` and JPEG, then with some new, lesser-known algorithms, such as lossless JPEG, `ha`, `codetree`, `progcode`, and `sp_compress`, and then with some semi-custom algorithms where we compress the hi-order bytes of the data separately from the lo-order bytes. We measure the efficiency of the methods in terms of the number of *bits per point* (bpp) in the compressed file, as well as the compression ratio relative to an uncompressed binary file (which is already smaller than an uncompressed ASCII file). We consider both lossless and lossy algorithms, and in the latter case measure the error compared to the bpp. We study the size penalty associated with partitioning the file into many small pieces before compression, which facilitates accessing only a small part of the file later without needing to uncompress all of it. We then try the best methods on 1024×1024 extracts from 24 randomly-selected $3''$ USGS DEMs and on 4 extracts from the ETOPO5 files.

Finally, we will consider some ways that a specifically topographic format, the TIN, might be compressed.

2 Review

This review will first consider data compression and entropy, then effects of errors in DEMS on interesting properties, and finally other operations on DEMs, which might be affected by errors.

For an excellent introduction to data compression in general, see the Usenet `comp.compression` Frequently Asked Questions file (FAQ)[12]. It discusses Huffman, Lempel-Ziv-Welch (LZW), JPEG, and others, and gives references. Nelson[23] is a good introduction to the standard compression algorithms, and includes floppies with code. Whitten[39] describes a suite of programs for text and image compression. One non-technical issue inhibiting optimal compression is the large number of patents on the most popular methods.

Burrough[1] and Peuquet[25] consider data compression issues in GIS. Fractals, as presented in Clarke and Schweizer[6], might also be used to compress terrain. Neumann[24] introduces information issues, such as entropy, in representing the topological relations in a map.

Carter[2] describes the errors in the $3''$ DEMs, produced by interpolating contours on 1:250K maps, which we use in this study. Walsh et al[36] also discuss this problem.

Weibel[38] filters gridded DEMs in various ways. He uses global filtering doing smoothing as in image processing by convolving with a 3×3 or 5×4 filter. He compares this with a selective filtering to eliminate points that do not add anything to our characterization of the surface. He tests a 220×390 elevation grid to see whether generalization changes essentials of the terrain, such as hill sharing and RMS error. Shea and McMaster[32] also discuss generalization.

Chang and Tsai[4] found that lowering DEM resolution hurt the accuracy of the calculated slope and aspect of the terrain. Carter[3] shows that the 1 meter resolution particularly affects the aspect, causing a bias towards the four cardinal directions, and suggests smoothing the data. Lee et al[19] analyze the effect of elevation errors on feature extraction. Fisher[10] considers the effect on visibility.

One operation often performed on terrain data is visibility determination, De Floriani and Magillo[7]. Puppo et al[26] use a parallel machine to convert a DEM to a TIN. They scale the elevation to 8 bits and perform experiments on grids of up to 512×512, reporting results for a 128×128 grid. For example, for a 30 meter accuracy 497 of the 16,384 points are selected. This TIN is then used to calculate line-of-sight-communication in De Floriani et al[8].

Drainage pattern determination is another frequent DEM operation, as described by

McCormack et al[22]. Skidmore[33] extracts properties of a location, such as being on a ridge line, from a DEM. Franklin and Ray[11, 27] do visibility calculations on large amounts of data.

The use of a linear quadtree with 2-D run-length encoding and Mortin sequences is discussed in Mark and Lauzon[21]. The storage can be about 7 bits per leaf. Waugh[37] critically evaluates when quadtrees are useful, while Chen and Tobler[5] find that quadtrees always require more space for a given accuracy than a ruled surface. Dutton[9] presents a region quadtree based on triangles, not squares. This quaternary triangular mesh defines coordinates on a quasi-spherical world better than a planar, Cartesian, system does. Leifer and Mark[20] use orthogonal polynomials of order up to 6 and quadtrees for a lossy compression of three 256×256 DEMs. Their work anticipates ideas used in wavelets and in the best methods that we will see later in this paper.

3 Compression Review

Here is a quick summary of some of the major themes in compression. *Huffman* coding measures the number of times each symbol in a file occurs, and uses fewer bits to represent the more common symbols. In English text, instead of using 7 bits for each character, *E* might be stored as *00*, which *Q* might be *10110010*. The limitation is that adjacencies, such as that *Q* is almost always followed by *U*, are not used. Huffman is optimal if there are no such relations, except for a roundoff error since each symbol must be coded with an integral number of bits even if its probability is not exactly an inverse power of two.

Arithmetic coding improves on Huffman in that case by coding symbols with real fractional numbers. Only enough bits to uniquely identify the symbol are stored. Arithmetic coding usually uses floating point math, and can be subject to roundoff errors, which will cause errors in the uncompressed file, unless care is taken in the implementation. Huffman and arithmetic coding are often used inside other schemes to compress calculated coefficients.

Run-length encoding represents strings of the same repeated symbol as a count. The *GIF* (Graphics Interchange Format) image format common on PCs uses this (and other techniques). This method is excellent for line drawings. A step more sophisticated than this is *delta* encoding, which calculates the difference of each pixel of an image from the previous pixel, and then uses some other method, such as run-length encoding, to compress the differences. Facsimile transmission does this in 2-D, calculating the difference of one row from the previous row, then within that row of differences, calculating second order differences across it. The 2-D differencing is about 10% better than 1-D.

The above two methods are *lossless*; the original file is completely reconstructible. For images, *lossy* methods are often used since they may allow much greater compression without much visibly hurting the image.

Wavelets[13] offer a new technique that appears quite powerful; some of the best compression systems described below are based on this.

JPEG, from its designers, the Joint Photographic Experts Group, is a standard image compression method that splits the image into blocks, does a discrete cosine transform frequency analysis of each block (similar to a Fourier transform), deletes components that should not be very visible, and Huffman codes some things. JPEG is usually lossy. It has a parameter to control how lossy, with a lossier file being smaller. JPEG was designed to compress hi-quality photographic images, with 24 bpp (3 colors at 8 bits). This contrasts to GIF, which assumes only 8 bpp. JPEG does not do so well compressing line drawings and text, with sharp jumps in intensity between adjacent pixels. There is also *LJPEG*, lossless JPEG. LJPEG may use various techniques, such as 2-D differencing, as well as a lossy JPEG followed by a compressed correction table. Huang and Smith[15] have an implementation that compresses binary (raw) PBM files. 2-D differencing also appears in other methods.

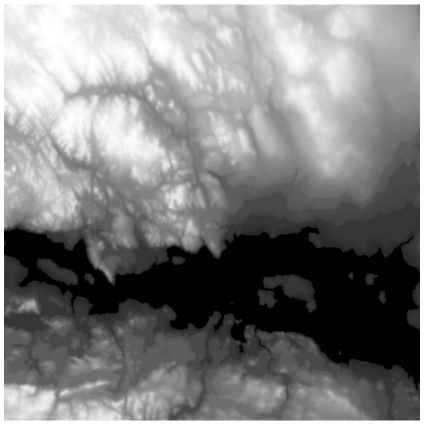
Fig. 1. Sample `Adir512`: Lake Champlain West DEM at 512×512

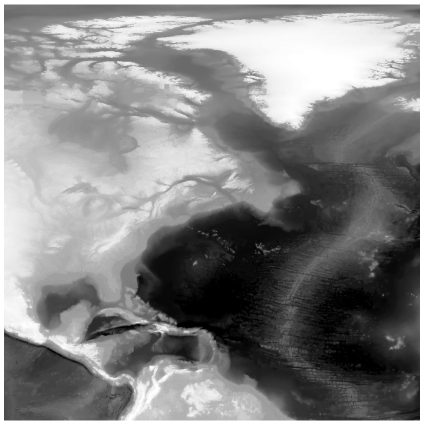
Fig. 2. 1024×1024 Section of ETOPO5

4 Lossless Compression Experiments

We started by experimenting with several lossless compression techniques on `adir512`, a 512×512 reduction of the USGS Lake Champlain West DEM, shown in Figure 1[1], obtained from the 1201×1201 DEM with the program `pnmscale`. This is not one corner of the DEM, but is the whole DEM at a lower resolution. We used this conservative choice since a 512×512 corner of the DEM, with the points still spaced at $3''$, would have compressed either better or worse depending on which corner we used, as shown later in Table 6 on page 14. The file contains a mix of a mountainous region in the southwest, with a plane in the center west and a lake in the center east. The $262,144$ points range in elevation from 22 to 1,568 meters.

The purpose of these experiments was to determine the relative performance of the various methods, so that we could select a few methods to apply to files of various sizes. The next test case was `adir1024`, a 1024×1024 corner of the Lake Champlain West DEM, first whole, then partitioned into smaller and smaller blocks. This demonstrated that altho a particular method might compress different files of the same size by amounts ranging over a factor of three, the relative performance of different methods applied to

[1] The printed greyscale image was histogram-equalized with `xv` to improve legibility.

the same file did not vary much. This is why these experiments might have some general validity on other data sets.

Finally we tested a few methods on a 1024×1024 block of 24 other randomly selected DEMs. We saw the same behavior again. Altho compression ratios varied over a factor of 10, the ratio of the size achieved by the `progcode` method to the size achieved by `gzip`, varied by only a factor of 2. That is,

$$\frac{\max_{i=1,24}\left(\frac{\text{size}(\texttt{progcode}(\text{file}_i))}{\text{size}(\texttt{gzip}(\text{file}_i))}\right)}{\min_{i=1,24}\left(\frac{\text{size}(\texttt{progcode}(\text{file}_i))}{\text{size}(\texttt{gzip}(\text{file}_i))}\right)} = 2.$$

In the following numbers, we will generally ignore header information in the files since the headers are a small fraction of the file size, which fraction gets smaller as the data gets larger.

The raw ASCII file, with each point stored as three to five characters, depending on the number of digits in the elevation (plus a separator), as takes 964,532 bytes. Using a binary file with two bytes per number reduces the space to 524,288 bytes. This is a smaller reduction than expected since 33.6% of the elevations are only 2 digits. Nevertheless, the binary file is useful since it is much faster to read since formatted I/O is rather compute-bound.

The Unix `compress` command, applied to the ASCII file, produces a 289,408 byte file, altho in only 3 CPU seconds on a Sun Sparc 10/30 workstation. A better compression program is `gzip`, part of the Free Software Foundation suite of free GNU software. `gzip` has an option for the amount of time spent to compress. At the default setting, applied to the original ASCII file, the compressed file is 263,270 bytes, produced in 21 seconds. `gzip -9`, the slowest smallest setting, produces a file of 258,016 bytes in 37 seconds. Since the latter took almost twice as long to compute, the default setting is probably appropriate.

`Gzip`, in 7 seconds, compresses the above binary file to 230,889 bytes. `Gzip -9` creates a slightly smaller 230,179 byte file in 14 seconds. This shows that treating the file as numbers, rather than as simple text is somewhat better. How good might we potentially do?

Among the 262,144 points, there are only 1,436 different elevations. Thus, a table of the existing elevations plus 10 bpp are almost sufficient. What is the information-theoretic Huffman lower bound, ignoring relations between adjacent points? Let n_i be the number of points with the i-th elevation, counting from lo to hi, and let the total number of points be $N = \sum n_i$. Then lower bound, also called the entropy of the data, is, in bits,

$$N \log_2 N - \sum_i n_i \log_2(n_i)$$

For this data, the frequencies range from 71 elevations occurring once, to one elevation occuring 28,674 times. The entropy is 280,753 bytes, or 8.6 bpp. Clearly, ignoring correlations in the elevations is expensive.

The above entropy calculation ignores autocorrelations in the data. Therefore consider a 1-D differencing, along a linear array of the points. There were 501 different differences ranging from –158 to +750, with 156 deltas occurring once, $\Delta = 0$ occurring

80,125 times, and the next most frequent delta occurring 15,577 times. The resulting file's entropy is 163,125 bytes, or 5.0 bpp, so using deltas gave us a factor of 1.7.

4.1 Ha

`Ha0.999`[14] is a recent (1995) compression program by Harri Hirvola designed for general files, either text or images. It has two methods, which are described in the documentation thus:

> *1-ASC:* Default method using sliding window dictionary followed by arithmetic coder. Offers quite good compression on wide variety of file types.
>
> *2-HSC:* Compression method based on finite context model and arithmetic coder. Quite slow for binary data, but offers very good compression especially for longer text files.

We tried both methods on both the ASCII and the binary heights. Compressing the binary data was, as usual, about 10% better. `Ha-2` was somewhat smaller than `ha-1`, and sometimes faster, sometimes slower. `Ha-2` on the binary data gave 3.2 bpp. However, `ha` is about twice as slow as `gzip`.

4.2 Splitting the Data Into Planes

Since the GIF format and many image processing implementations require 8-bit data, we might split our 16-bit data into two separate 8-bit planes, containing the hi-order and lo-order bytes, respectively, of each elevation, and encode each plane separately. Of three different compression algorithms applied to the hi file, GIF was the best, somewhat better than `gzip` on the binary file. `Gzip` was slightly the best on the low file, but all the method were close enough that the difference is probably not significant.

Combining GIF on the hi with `gzip` on the binary lo gave 191,670 bytes, or 5.9 bpp, or less efficient than some following methods.

Splitting also illustrates a weakness in `gzip`. It compressed the 1-byte-per-point lo file down to 182,038 bytes. However, we tried expanding the lo file to 2 bytes per point, with the hi byte always zero. `Gzip` compressed this only to 217,335 bytes, 20% worse. This also occurred when the hi file was expanded to 2 bytes per point. `Gzip`'s compressed size grew from 9.711 to 11,433 bytes. Also `gzip` on the two halves of the file was more efficient than `gzip` on the whole file. This was true for the file in either ASCII or binary. In the latter case, the difference was 17%.

4.3 Said and Pearlman

Said and Pearlman[28–31] have several new lossless and lossy image compression algorithms and programs. The uncompression programs are separate in order to facilitate validating the results. `Sp_compress` is a lossless progressive transmission method using arithmetic coding. Progressive transmission means that lo-resolution data is transmitted first in the file, then it is refined. This is useful since an uncompresser could be written for previewing a file via a lo-speed communication medium before deciding whether to get the whole file. `Sp_compress` compressed `adir512` to 96,413 bytes, or 2.94 bpp, in 5 seconds.

`Progcode` is another lossless compression program, which produces a slightly larger file. Its decoder, `progdecd`, altho not written to uncompress a truncated file, calculates what the mean-squared error (MSE) would be if a lower bit rate were used. The MSE shows that `progcode` is quite conservative with the compressed file size, presumably to prevent possible roundoff errors similar to what can occur with arithmetic coding. For example, `progcode` compressed `adir512` to 98,993 bytes, or 3 bpp, in 6 seconds. However when this compressed file is decoded at a rate of only 1.2 bpp, `progcode` reported that the result was identical to the original file. This means that `progcode` could presumably be modified losslessly to compress this file to only 32,322 bytes, which is a very high compression ratio. In our compression experiments, after running `progcode` on a file, we often ran `progdecd` at various bit rates to find the smallest rate at which it reported no differences in the uncompressed file. We name this number the `progcode` *extrapolated* rate to indicate that we expect that this rate might be possible with a modified `progcode`, but that we don't yet have a program that does it.

The obvious question is whether the low `progcode` extrapolations might be due to some error. However the Said and Pearlman program `codetree`, described later in the lossy compression section, suggests that the `progcode` extrapolations are correct. When using `codetree`, we give it a bit rate, and it writes a file of the corresponding size. Then the separate uncompression program `decdtree` uncompresses that file, compares it to the original, and reports the mean-squared error.

In this example, `codetree` at 1.2 bpp compressed the file to 39,322 bytes. `Decdtree` at that bit rate reported a mean-squared error of 1.45, where the maximum elevation was 1,568. We also converted the uncompressed file to PGM format, differenced it with the original, and calculated the histogram of the absolute differences between the original file and the 1.2 bit per point compression, as shown in Table 1 on the facing page.

Table 1. Histogram of Errors Caused by Compressing `Adir512` With `Codetree` at 1.2 Bpp

Absolute Difference	Number of Points
0	111,725
1	102,975
2	34,851
3	9,794
4	2,301
5	425
6	61
7	11
8	1

Said and Pearlman's `sp_compress` and `progcode` are by far the best. `Ha-2` on the binary file is second, far behind, with `gzip` well behind it. Splitting the file into hi and lo bytes and compressing them with GIF and `gzip` respectively was better than only `gzip`. Two conclusions are evident.

1. Generic compression algorithms, usually in the image processing domain, are so good that there may be no need to design special algorithms for Digital Elevation Models.

2. The two best algorithms are less than two years old, which shows that compression research is still progressing.

`Progcode` has one restriction: the size of the data array must be a power of two. This poses a problem with DEMs, which have 20% more than a power of two rows. The easy solution, for the programmer, is to let the user repartition his data to suit the program, but a better solution might be to pad the DEM to the next power of two, 2048×2048, which increases the total size by a factor of 2.9.

How efficiently does `progcode` compress a file with large blocks of zeros? We made two tests of compressing a file, then padding it with zeros to double its size, and compressing that. The first file was `adir512`; the second `adir1024`. The latter was statistically a little different in that it was a subsquare of the DEM, and not a subsample of the whole, as was the former. As Table 2 on the next page shows, the compressed padded files were each 16% larger than the corresponding compressed unpadded file. There are some internal parameters in `progcode`, which are now set for medical images. If they were fine-tuned for the statistical properties of elevation data with large blocks of zeros, then this 16% padding penalty might be reduced. Alternatively, instead of padding the file up, it might be partitioned, quadtree-style, into smaller blocks each of an acceptable size. At some minimum size, such as 8×8 perhaps, we might stop splitting and start zero-padding. Then, since there is little overhead in compressing small blocks, each block could be feasibly separately compressed.

Since the actual size of the compressed file actually needed by `progcode` for error-free compression seems to be much smaller than the whole file, we considered that for the first test case. However, the results were comparable. The 512 file needed 1.2 bpp for error-free reconstruction, while the padded 1024 file need 0.35 bits per padded point, or 1.4 bits per original point, or about 16% more.

`Sp_compress` requires only that the number of rows and columns each be a multiple of four so the padding problem does not arise here. We reported above how well it handles padded files since this is one measure of its optimality.

4.4 Summary

Table 3 on page 11 summarizes the compression algorithms that don't split the data into planes. We conservatively calculated the compression ratio relative to the binary file size, not to the original ASCII file. Table 4 on page 12 shows the results when the elevations were first split into hi and lo order bytes, and the two files compressed separately. Showing the compression ratios for the separate bytes didn't seem meaningful.

Table 2. Compression Efficiency on Zero-Padded DEMs

	`Adir512`		`Adir1024`	
	Original	Padded	Original	Padded
Number of rows	512	1024	1024	2048
Progcode:				
Compressed file size	98,993	113,641	254,952	295,857
Ratio	1.16		1.16	
Bits per original point	3.0	3.5	1.95	2.26
CPU Time	5	15	25	82
Sp_compress:				
Compressed file size	96,413	103,884	246,923	275,327
Ratio	1.08		1.11	
Bits per original point	2.9	3.2	1.9	2.1
CPU Time	5	15	21	62

Finally, in these tests, we used Sun Sparc IPC and 10/30 Unix workstations, and programmed in C++ with the Apogee compiler. Tomas Rokicki's `dvips`, Jef Poskanzer's Portable Bit Map (PBM) package, Netpbm version, and John Bradley's `xv` were also very useful.

5 Lossy Compression Experiments

Since elevation data can have many errors[2], and in any case is represented only to the nearest meter, it is a waste to store the data at a higher accuracy than is justified. Therefore *lossy* compression methods might be appropriate sometimes. Indeed, sometimes we gather data faster than we can easily store or transmit it. Altho the our first suggestion here might be simply to lower the spatial resolution or to truncate the number of bits used to store each point, a more sophisticated approach, such as one of the algorithms described below, might allow us to keep more information in fewer bpp.

5.1 Scaling the Elevations

There are a large number of image processing compression methods. Altho their underlying theory is general, many implementations are for only 8-bit data. There is also some precedent in the literature for scaling elevations to 8 bits for research purposes. In `adir512`, scaling destroys $\log_2\left(\frac{1568}{256}\right)$, or slightly more than 2 bpp, or 65K bytes. The resulting file is now quantized in 6 meter intervals, so the maximum error introduced by this is 3 meters, and the average error is 1.5. Datasets with a greater maximum elevation will have larger errors.

After scaling the elevations, the binary file is one byte per element, or 262,144 bytes. `Gzip` compresses this file to 103,413 bytes. Note that this size, plus the destroyed bits, is still less than the `gzip`ped original file, showing in yet another way that altho `gzip`

Table 3. Comparison of Various Compression Methods on `Adir512`

Method	*File Size*	*Time*	*Ratio*	*Bpp*
	(A)		$\left(\frac{524{,}288}{A}\right)$	$\left(\frac{A}{32{,}768}\right)$
Lossless:				
Original ASCII file	964,532		0.54	29.4
Binary file	524,288		**1.0**	16.0
`Compress`	289,408	3	1.8	8.8
Entropy of the elevations	280,753		1.9	8.6
`Gzip` on the ASCII file	263,270	21	2.0	8.0
`Gzip-9` on ASCII	258,016	37	2.0	7.9
`Ha-1` on ASCII	256,396	40	2.1	7.8
`Gzip` on binary	230,899	7	2.3	7.0
`Gzip-9` on binary	230,179	14	2.3	7.0
`Ha-1` on binary	224,904	18	2.3	6.9
GIF on hi and `gzip`-bin on lo	191,670		2.7	5.9
`Ha-2` on ASCII	179,146	30	2.9	5.5
Entropy of the deltas	163,125		3.2	5.0
`Ha-2` on binary	163,115	38	3.2	5.0
`Progcode`	98,993	6	5.3	3.0
`Sp_compress`	96,413	5	5.5	2.9
Extrapolated `Progcode`	32,322		16.4	1.0
Scaled down; otherwise lossless:				
Binary file	262,144		2.0	8
GIF	123,986		4.2	3.8
`Gzip`	103,413		2.0	3.2
Lossless JPEG	64,935		8.1	2.0
Scaled down and lossy:				
JPEG 75	14,798		35	0.45
JPEG 50	10,408		50	0.32
JPEG 25	7,275		72	0.22

is an excellent compression program, it can be suboptimal.

Converting the scaled file to a GIF file (which is exact, but requires 8-bit data) gives a 123,986 byte result, somewhat worse than `gzip`. LJPEG gives 64,935 bytes, showing that elevation data has similar statistical characteristics to photographic images. This may be partly due to smoothing during the data generation, in which cliffs in the real world can be smoothed into gradual slopes.

Table 4. Comparison of Compression Methods on the `Adir512` Split into Hi and Lo Bytes

Method	*File Size*	*Ratio*	*Bpp*
	(A)	$\left(\frac{262,144}{A}\right)$	$\left(\frac{A}{32,768}\right)$
Hi bytes:			
Original	262,144	1	8
`Gzip` on ASCII	11,393	24	0.35
`Gzip` on binary	9,711	27	0.30
GIF	9,632	27	0.29
Lo bytes:			
Original	262,144	1	8
`Gzip` on ASCII	226,246	1.15	6.9
`Gzip` on binary	182,038	1.4	5.6
GIF	227,833	1.15	7.0
Combined:	(A)	$\left(\frac{524,288}{A}\right)$	$\left(\frac{A}{32,768}\right)$
Original	524,288	1	16
`Gzip` on each ASCII file	237,639	2.2	7.3
GIF on each file	237,465	2.2	7.3
GIF on hi and `gzip`-binary on lo	191,670	2.7	5.9

5.2 Lossy Compression of the Scaled Data

JPEG compression, by the `xv` program, at the default parameter setting of 75%, gives a 14,798 byte file. (The JPEG parameter scale is arbitrary, and does not mean, say, 75% accuracy.) Parameters of 50 and 25 give files of 10,408 and 7,275 bytes respectively. The compression is quite impressive, if you can tolerate the errors. Table 5 on the facing page shows a histogram of the errors for three JPEG parameter setting. For the 75 parameter, $1/2$ of the elevations were not changed by JPEG compression, the average elevation was changed by 0.626 scaled units, which is 4 original units (meters), and the worst change was 8 scaled units. The RMS error is 1.015 scaled units, 6.2 original units.

`xv` also has a smoothing parameter for JPEG compression. However it doesn't have much effect; setting it to the high value of 50% reduced the compressed files by about 2%.

5.3 Said and Pearlman

Said and Pearlman also have two lossy compression programs, `Codetree` and `fastcode`, using wavelet compression. `codetree` adds arithmetic entropy-coding, which improves the compression a little, but increases the time a lot. With these programs, you specify the desired number of bits per pixel, and whether the input data is

Table 5. Histogram of Errors Caused by Various JPEG Parameter Settings on the Scaled `Adir512`

Absolute Error	*JPEG 75*	*Frequency* *JPEG 50*	*JPEG 25*
0	138,263	68,016	38,916
1	93,265	132,363	112,120
2	23,265	38,104	53,840
3	5,597	14,512	27,545
4	1,321	5,616	14,294
5	337	2,196	7,157
6	79	860	3,864
7	14	314	2,120
8	3	102	1,139
9		42	561
10		15	265
11		4	159
12			76
13			41
14			28
15			10
16			6
17			2
18			0
19			1
File Size	14,798	10.408	7,275
Bpp	0.45	0.32	0.22

smooth (used for optimizing the compression). However, since `progcode` also gives almost as good lossy compression, by decoding only part of the compressed file, we used that. Figure 3 on the next page shows the RMS (Root-Mean-Square) error of the compressed file for sizes ranging from 0.025 to 1.2 bpp, at which bit rate the reconstructed file is identical to the original.

We can now compare `prograte` to JPEG on the scaled data. For example, JPEG-75 has 0.45 bpp, and an RMS error of 6.2. However, `prograte`'s RMS error at this reconstruction rate is 3.9. `Codetree`'s RMS error at 0.45 bpp is 3.5. Alternatively, these programs can achieve an RMS error of 6.2 with a coding rate of only 0.25 bpp. Therefore they are almost twice as efficient as JPEG on the scaled file.

6 Compressing the DEM in Pieces

One objection to compressing data is that using any part of the file requires uncompressing the whole file. We could partition the file into pieces and compress them separately, but then the total size of all the compressed pieces might be larger than the compressed single file. To test this, we split `adir1024` into four 512×512 pieces. Table 6 on

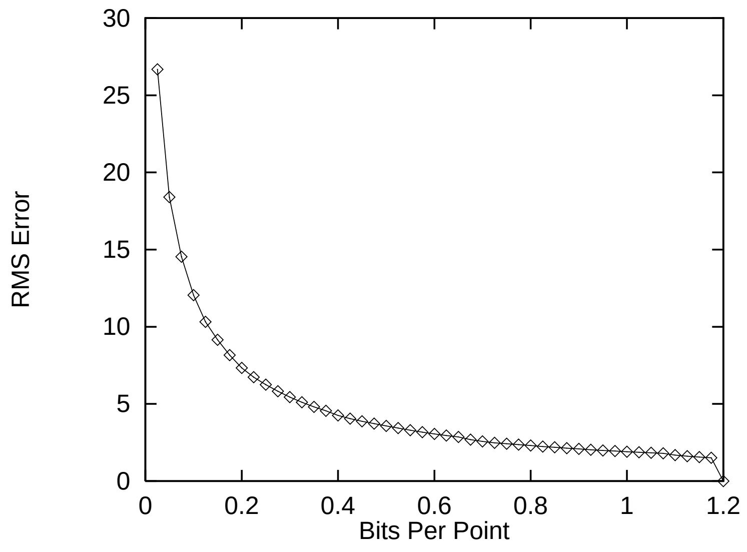

Fig. 3. Lossy Compression `Adir512`

Table 6. Effect of Quartering `Adir1024` Before Compressing

Method	*Compressed File Size*	*Compressed Quarters Sizes*	*Ratio*
`Gzip`	757,013	307137+208268+144059+75078=734,542	97.0%
`Sp_compress`	246,923	98463+63387+53490+32239=247,579	100.3%
`Progcode`	254,952	100400+64831+55301+33477=254,009	99.6%

the following page shows that quartering the file has little effect, but sometimes slightly improves the compression. While this might not always obtain, it suggests that splitting the file before compressing should not be disastrous.

This table also suggests how variable compression can be. With either method the largest compressed quarter was triple the size of the smallest. This could cause a problem in practice since a program handling compressed files would either have to dynamically allocate storage according to each file's size, or else always have enough storage reserved for the rare worst case.

What if we split `adir1024` into 16 pieces of 256×256 and compress them separately? `Gzip` gives files ranging from 12,684 to 81,199, for a total size of 731,067 bytes, still smaller than the compressed complete file. The `sp_compress`ed files range from 6,386 to 27,098 bytes, for a total of 251,042 bytes, or only 1.7% larger than the original compressed file. What about partitioning `adir1024` into 64 blocks of 128×128? The `gzip`ped files range from 357 to 21,739 bytes, totaling 747,438 bytes. With `sp_compress`, the files range from 279 to 7,455 bytes, totaling 258,489 bytes.

Table 7. Effect of Partitioning `Adir1024` Into Blocks Before Compressing

Ratio is the ratio of the total of the sizes of all the compressed blocks to the size of `Adir1204` compressed as one file.

Method	*Number of Blocks*	*Block Size*	*Ratio*
`Gzip`	1	1024	1.00
	4	512	0.97
	16	256	0.97
	64	128	0.99
	256	64	1.02
`Sp_compress`	1	1024	1.00
	4	512	1.00
	16	256	1.02
	64	128	1.05
	256	64	1.13
`Progcode`	1	1024	1.00
	4	512	1.00

What about 256 blocks of 64×64? `Gzip` produced files from 45 to 5,827 bytes, totaling 771,455 bytes. `sp_compress` crashed while compressing three of the files, which were almost all zeros, so we used `progcode` for them. The 256 files ranged from 27 to 2,217 bytes, totaling 279,060 bytes. Table 7 summarizes these results, which may be stated briefly thus: Compression is useful even when we wish to use only a portion of the file.

7 Large Scale Test of 24 DEM Samples

In the above section, it seems that `progcode` is the best lossless compression of elevation data. Does this extend to other data? The full test was to take a random sample of all the USGS DEMs[35] (except Alaska because of its different size). For each letter of the alphabet (except X and Z), we took the alphabetically first available DEM starting with that letter. They are shown in Figure 4 on the following page. We losslessly compressed a 1024×1024 extract from each file thrice, with `gzip` (for comparison since it is so popular), `progcode`, and `sp_compress`.

Table 8 on page 17 shows the resulting number of bpp, and the compression ratio, measured relative to the binary file size. We report the arithmetic average, not the, lower, geometric mean at that bottom since this is more conservative and also more relevant when compressing and storing many files at once.

Note that both the `gzip`ped and our compressed sizes vary wildly from file to file. Each original file was $2 \cdot 1024^2 = 2,097,152$ bytes. All the compression times were in the range from 12 to 26 seconds. In every case, `sp_compress` and `progcode` were better than `gzip`, compressing the whole set of 24 files down to 2.0 and 2.1 bpp, respec-

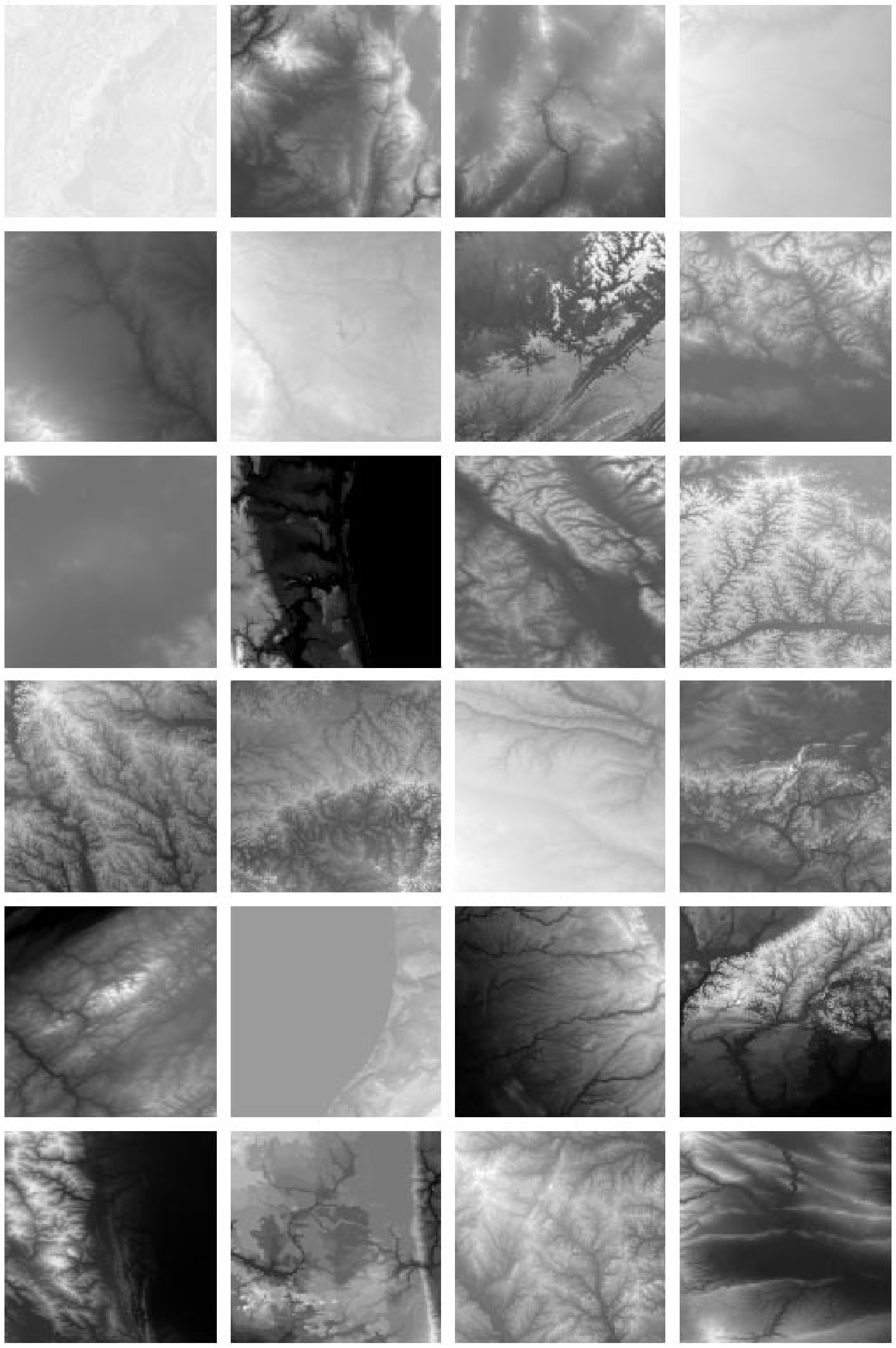

Fig. 4. The 24 Sample USGS DEMs

Table 8. `Gzip`, `Progcode`, and `Sp_compress` Compressing 24 Random USGS DEMs

File	*Gzipped Binary File Size*	*Progcode File Size*	*Progcode Compression Ratio*	*Progcode Bpp*	*Sp_compress Bpp*
		(A)	$\left(\frac{2097152}{A}\right)$	$\left(\frac{8A}{1024^2}\right)$	
Aberdeen E	159,605	119,465	17	0.9	0.88
Baker E	1,003,125	448,446	4	3.4	3.32
Caliente E	935,762	395,299	5	3.0	2.93
Dalhart E	274,551	170,611	12	1.3	1.27
Eagle Pass E	360,627	193,790	10	1.5	1.46
Fairmont E	260,759	170,128	12	1.3	1.27
Gadsden E	620,059	312,823	6	2.4	2.29
Hailey E	1,082,799	425,666	4	3.3	3.15
Idaho Falls E	286,591	174,438	12	1.3	1.31
Jacksonville W	103,181	79,852	26	0.6	0.58
Kalispell E	993,627	472,969	4	3.6	3.52
La Crosse E	795,570	472,465	4	3.6	3.51
Macon E	374,493	219,867	9	1.7	1.62
Nashville E	566,738	285,661	7	2.2	2.11
O'Neill E	379,707	208,384	10	1.6	1.55
Paducah E	440,521	235,205	8	1.8	1.74
Quebec E	660,238	282,027	7	2.2	2.09
Racine E	73,776	50,908	41	0.4	0.36
Sacramento E	1,014,075	516,623	4	3.9	3.85
Tallahassee E	309,067	192,788	10	1.5	1.42
Ukiah E	927,052	460,870	4	3.5	3.45
Valdosta E	157,818	118,538	17	0.9	0.88
Waco E	346,640	203,288	10	1.5	1.52
Yakima E	898,159	353,717	5	2.7	2.61
Total	13,024,540	6,563,828	7.7	2.1	2.0

tively, or half the size of `gzip`. Therefore, both `sp_compress` and `progcode` seem excellent choices for compressing gridded elevation data.

Testing `progdecd` at different bit rates on the files compressed by `progcode` showed that on the average there were no differences at an uncompression rate of 0.7 bpp. This shows us a goal for lossless compression.

8 Compressing ETOPO5

The *ETOPO5* file for the northern hemisphere is an array of 1080×4320 elevations or depths, spaced at $5'$ of arc. We extracted 4 1024×1024 subsets, starting 56 rows down from the North Pole, and starting at the first column, and at multiples of 1024 columns later. Fig. 2 on page 5 shows the fourth section. Then we tried `gzip`, `progcode`, and `sp_compress` on them, with the results shown in Table 9. `sp_compress` looped on section 2, so we used `progcode` for that one. This data does not compress so much since the elevation range is much greater (about 19,000). `Sp_compress` or `progcode` are still better than `gzip`.

Table 9. Comparison of Various Compression Methods on ETOPO5 Sections

Method	*Section*	*Size*	*Time*	*Ratio*	*Bpp*
		(A)		$\left(\frac{2097152}{A}\right)$	$\left(\frac{8A}{1024^2}\right)$
Original file	1-4	2,097,152		**1.0**	16
`Gzip`	1	1,011,838	14	2.1	7.7
	2	1,188,712	19	1.8	9.1
	3	1,413,901	20	1.5	10.7
	4	1,367,549	19	1.5	10.4
	Average	1,245,500		1.7	9.5
`Sp_compress`	1	726,955	27	2.9	5.5
	2	*failed*			
	3	829,678	29	2.5	6.3
	4	882,405	19	2.4	6.7
`Progcode`	2	819,326	33	2.6	6.3
	Average	814,591		2.6	6.2

9 Compressing a TIN

Altho regularly gridded data compresses quite well, how might we compress a TIN file? There are the (x, y, z) coordinates of the points, and the six neighbors (on average) of each point.

Let the number of points in the original DEM be N, typically, $1201^2 = 1,442,401$. Let K be the number of points selected to be in the TIN. If the TIN is to be worth using, then $K \ll N$. The easiest way to store the points is to list their coordinates in the original DEM, at a cost for each (x, y) of $2 \log_2 N$ bits/point. However, the smaller information-theoretical space comes from the number of subsets of K points selected from N, and is

$$\frac{1}{K} \log_2 \mathrm{binom}(N, K) \approx \lg N - \lg K$$

bits per point, where $\mathrm{binom}(N, K)$ counts binomial combinations, e.g., $\mathrm{binom}(5, 2) = 5!/2!/(5-2)! = 10$. If, say, $K = 65,535$, then this is a significant reduction from 20.5 to 4.5 bits per point.

Compressing heights is hard since the irregular topology makes run-length encoding difficult, tho not necessarily impossible. Compressing the topology is harder. If there is nothing stored inside each triangle, then they might be constructed as needed from the point adjacencies.

The average point has six neighbors, and the simplest implementation is an array of the ID numbers of the neighbors. 16 bits per ID, plus a count, gives about 100 bits per point. This is by far the dominant storage cost. Note that the storage cost for a hierarchical structure will be even larger. There are more compact methods of storing a planar graph, so that each edge requires only a few bits. The compact format has to be expanded before the graph can be traversed, but that's already the case with the points. To understand the compact planar graph structure, consider how to store a binary tree compactly.

With the simple binary tree format, each node contains two pointers, for its two sons. We can traverse and update this structure, but it costs 32 bits per node (if there are under 2^{16} nodes). We might instead store only the information about whether or not each node has a left son, and whether it has a right son, in 2 bits total per node. We traverse the tree in some well-defined recursive order, such as the node, then its left subtree, and finally its right subtree, listing the 2 bits for each node in sequence. Following that, we can list in sequence any internal information about each node.

We can extend this to a planar graph, though the structure is more complicated; see Jacobson[16, 17], Kannan et al[18], and Turan[34]. An unlabeled triangulation can be stored in $3 + \lg 3 \approx 4.6$ bits per node; the K labels require another $K \lg K$. The graph would probably need to be expanded before being used.

Altho the TIN initially does not seem competitive with the regular grid compression methods described earlier, with these succinct codings, it might be. Which is actually better remains an open question.

10 Summary

We have studied many compression algorithms for regularly gridded terrain elevation files, including both generic image processing methods, and some semicustom ones. The generic image processing compression algorithms perform so well that there appears no need to design algorithms specifically for elevation data. This also allows us to take advantage of the continuing progress in image processing algorithms. For example, the best algorithms that we studied are all quite new. With `sp_compress`, for example, USGS DEMs compress down to an average of 2 bpp. There is a wide variation in the compressability of different data; however the relative performance of various methods on any particular file does not vary widely.

Compression should not hinder interactive use of the data; partitioning one test file into 256 blocks before compressing increased thae total size by only 13%.

Several open questions remain. Can these methods be improved by fine-tuning internal parameters for the statistical properties of elevation data, if those properties are, in fact, different than the scenes that the algorithms were designed for? Can the very low bit-rates at which `progdecd` reports an exact reconstruction be extended into a new compression program at those rates? Since lossy compression is much more compact, often far below 1 bpp at moderate mean squared errors, how much lossiness can we tolerate before essential properties of the data such as drainage patterns and visibility are damaged?

References

[1] P. A. Burrough. *Principles of Geographical Information Systems for Land Resources Assessment*. Clarendon Press, Oxford, 1986.

[2] J. R. Carter. Relative errors identified in USGS gridded DEMs. In *Autocarto*, volume 9, pages 255–265, 1989.

[3] J. R. Carter. The effect of data precision on the calculation of slope and aspect using gridded DEMs. *Cartographica*, 29(1):22–34, Spring 1992.

[4] K.-T. Chang and B.-W. Tsai. The effect of DEM resolution on slope and aspect mapping. *Cartography and Geographic Information Systems*, 18(1):69–77, 1991.

[5] Z.-T. Chen and W. Tobler. Quadtree representations of digital terrain. In *Proceedings, Auto-Carto London*, volume 1, 1986.

[6] K. C. Clarke and D. M. Schweizer. Measuring the fractal dimension of natural surfaces using a robust fractal estimator. *Cartography and Geographic Information Systems*, 18(1):37–47, 1991.

[7] L. De Floriani and P. Magillo. Visibility algorithms on DTMs. *Int. J. Geographic Information Systems*, 8(1):13–41, 1994.

[8] L. De Floriani, P. Magillo, and E. Puppo. Line of sight communication on terrain models. *Int. J. Geographic Information Systems*, 8(4):329–342, 1994.

[9] G. Dutton. Locational properties of quaternary triangular meshes. In K. Brassel and H. Kishimoto, editors, *4th International Symposium on Spatial Data Handling*, volume 2, pages 901–910, Zürich, 23-27 July 1990.

[10] P. F. Fisher. Algorithm and implementation uncertainty in viewshed analysis. *International Journal Of Geographical Information Systems*, 7:331–347, Jul–Aug 1993.

[11] W. R. Franklin and C. Ray. Higher isn't necessarily better: Visibility algorithms and experiments. In T. C. Waugh and R. G. Healey, editors, *Advances in GIS Research: Sixth International Symposium on Spatial Data Handling*, pages 751–770, Edinburgh, 5–9 Sept 1994. Taylor & Francis.

[12] J.-L. Gailly. Comp.compression Frequently Asked Questions. Usenet, posted to comp.compression, ftpable from rtfm.mit.edu:/pub/usenet/news.answers/compression-faq/part[1-3], webbable from http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/top.html, 25 Feb 1995.

[13] J.-L. Gailly. What is wavelet theory? http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/part2/faq-doc-3.html, 25 Feb 1995.

[14] H. Hirvola. (HA, a small file archiver utility). ftp://ftp.nl.net/gopher/NLnet-connected/aipnl/ha_src/, Jan. 1995. Mentioned in the comp.compression FAQ.

[15] K. Huang and B. Smith. Lossless JPEG codec. ftp://ftp.cs.cornell.edu:/pub/multimed/ljpg.tar.Z, June 1994.

[16] G. Jacobson. *Foundations of Computer Science*, volume 30, chapter Space-efficient static trees and graphs. 1989.

[17] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie-Mellon, Jan. 1989. Tech Rep CMU-CS-89-112.

[18] S. Kannan, M. Naor, and S. Rudich. Implicit representation of graphs. *SIAM Journal On Discrete Mathematics*, 5:596–603, Nov. 1992.

[19] J. Lee, P. K. Snyder, and P. F. Fisher. Modeling the effect of data errors on feature extraction from digital elevation models. *Photogrammetric Engineering And Remote Sensing*, 58:1461–1467, Oct. 1993.

[20] L. A. Leifer and D. M. Mark. Recursive approximation of topographic data using quadtrees and orthogonal polynomials. In N. R. Chrisman, editor, *Autocarto 8: Proceedings Eighth International Symposium on Computer-Assisted Cartography*, pages 650–659, Baltimore, 29 March – 3 April 1987. ASPRS and ACSM.

[21] D. M. Mark and J. P. Lauzon. Linear quadtrees for geographic information systems. In *Proceedings of the International Symposium on Spatial Data Handling*, volume 2, pages 412–430, Zurich, 20–24 August 1984.

[22] J. E. McCormack, M. N. Gahegan, S. A. Roberts, J. Hogy, and B. S. Hoyle. Feature-based derivation of drainage networks. *Int. J. Geographic Information Systems*, 7(3):263–279, 1993.

[23] M. Nelson. *The Data Compression Handbook*. M&T Books, Redwood City, Calif. USA, 1991.

[24] J. Neumann. The topological information context of a map/ an attempt at a rehabilitation of information theory in cartography. *Cartographica*, 31(1):26–34, Spring 1994.

[25] D. J. Peuquet. A hybrid structure for the storage and manipulation of Very Large Spatial Data Structures. *Computer Vision, Graphics, and Image Processing*, 24(14), 1983.

[26] E. Puppo, L. Davis, D. de Menthon, and Y. A. Teng. Parallel terrain triangulation. *Int. J. Geographic Information Systems*, 8(2):105–128, 1994.

[27] C. K. Ray. *Representing Visibility for Siting Problems*. PhD thesis, Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, May 1994.

[28] A. Said. An image multiresolution representation for lossless and lossy compression. (submitted), July 1994.

[29] A. Said and W. A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. (submitted), presented in part at the IEEE Symp on Circuits and Systems, Chicago, May 1993.

[30] A. Said and W. A. Pearlman. Reversible image compression via multiresolution representation and predictive coding. In *Proceedings SPIE*, volume 2094: Visual Commun. and Image Processing, pages 664–674, Nov. 1993. email: amir@densis.fee.unicamp.br, pearlman@ecse.rpi.edu.

[31] A. Said and W. A. Pearlman. (new image coding and decoding programs). ftp://ftp.ipl.rpi.edu/pub/-EW_Code/, Apr. 1995.

[32] K. S. Shea and R. B. McMaster. Cartographic generalization in a digital environment: When and how to generalize. In *Autocarto*, volume 9, pages 56–67, 1989.

[33] A. K. Skidmore. Terrain position as mapped from a gridded digital elevation model. *Int. J. Geographic Information Systems*, 4(1):33–49, 1990.

[34] G. Turan. Succinct representations of graphs. *Discrete Applied Math*, 8:289–294, 1984.

[35] USGS. 1:250K DEMs. ftp://edcftp.cr.usgs.gov/pub/data/DEM/250.

[36] S. J. Walsh, D. R. Lightfoot, and D. R. Butler. Recognition and assessment of error in geographic information systems. *Photogrammetry Engineering and Remote Sensing*, 53:1423–1430, 1987.

[37] T. Waugh. A response to recent papers and articles on the use of quadtrees for geographic information systems. In *Proceedings of the Second International Symposium on Geographic Information Systems*, pages 33–37, Seattle, Wash. USA, 5–10 July 1986.

[38] R. Weibel. Models and experiments for adaptive computer-assisted terrain generalization. *Cartography and Geographic Information Systems*, 19(3):133–153, 1992.

[39] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, 1994.