The slim data compression system: algorithms and file format

Joe Fowler

October 23, 2008

Abstract

The program `slim` performs lossless compression on binary data files. It is designed to operate very rapidly and achieve better compression on noisy physics data than general-purpose tools designed primarily for text. `slim` is effective because it uses the facts that data from scientific instruments tend to be recorded in multiple-byte words and that successive words tend to be very close to one another rather than sampling from the full range of possibilities. This article describes the file format for slimmed files and the algorithms involved in slimming and unslimming data.

1 Purpose and approach of the `slim` program

The `slim` system performs lossless compression on binary data files and also re-expansion. It consists of both executable programs and a library with C linkage. `slim` is designed to operate very rapidly and achieve better compression on noisy physics data than general-purpose tools such as `gzip` and `bzip2`. By "noisy physics data" we mean data generated by real-world measurement systems with analog-to-digital converters (ADCs). Such data typically has the property that any given ADC will generate values from only a small fraction of the theoretically available range. This inefficiency might result from the ADC being stored for convenience in a wider word than necessary (as a 12-bit value being stored in a 16-bit word), or it might result from the physical input staying nearly constant (as a 1 to 300 K thermometer might when the temperature is held near 4 K for hours at a time). `slim` also works well in the related situation where the ADC channel takes on a wide range of values but changes only slowly. In this case, the difference between successive values has the useful property of being generally close to zero and `slim` can encode successive differences instead of the values themselves. In this and other `slim` documentation, we use "deltas" as a shorthand for the differences between successive data values.

In either situation, the data words will tend to have high-order bits that never change, low-order bits that are more or less uniformly distributed among all possible values,[1] and a few middle bits that have a correlated and non-uniform distribution. The approach `slim` takes to data is to subtract a single number, the "pedestal," from all the data recorded by one channel, which will make a certain number of high-order bits always (or nearly always) be zero. It is then safe to record only the n middle and low order bits.

Data from instruments can have occasional pathologies, so `slim` encodings are always universal codes. No matter how unlikely a data value might seem based on the past, `slim`

[1] For the special case of instruments that leave a few low-order bits strictly constant, see Section 5 about "bit rotation."

can always encode that value, though of course the unlikely value will require more bits than a likely value requires. The approach used here is to reserve one of the 2^n possible values to signal an overflow.

Because of the universal coding system, `slim` has the advantage that it can determine the near-optimal pedestal and number of bits for a data set without having to ensure that it can encode *every* value under such a system. Indeed, it does not even need to check every value in advance. Instead, `slim` looks at a fraction of the data (usually around 10%) and chooses the optimal pedestal and number of bits for that sample of the data. This sampling technique speeds up the compression step with almost no loss in compression ratio (unless one gets *very* unlucky about what is happening in the untested 90% of the data).

The discussion so far has equated one ADC channel with one file. Although this is the simplest case, `slim` can mix multiple channels in one file, provided that the file is structured in a well-defined repeating pattern of channels. Section 2 describes the allowed arrangement of data channels in a raw file and the terminology used for the various structured components of the raw file.

Section 3 specifies the details of the slimmed data file. Section 4 describes the compression algorithms used by `slim` and the word-level details of how they encode values. Some bit-level details are further explained in Appendix A. Appendix B discusses some compression algorithms that have been explored but are not used in `slim`, generally owing to their relatively expensive yet marginal gains in compression ratio.

2 Allowed arrangements of the raw data files and terminology

Raw data files, also called "fatfiles," are the targets of `slim`, whose job is to convert them to "slimfiles." Although `slim` can convert any fatfile to a slimfile, the conversion will work as an effective compression only in rather specialized cases. Fortunately, such cases are common in noisy physics instrument data, so long as the raw data file is treated according to the *meaning* of its bytes. Notice that `slim` will not compress data effectively unless it understands the contents of the input fatfile. In particular, it needs to know how to identify all the data words generated by the same instrument. This section explains the required arrangement of the fatfile and the language used in this document (and in the `slim` source code) for the analysis of a fatfile.

Section The fatfile consists of one or more *sections*. Sections are treated as having potentially distinct statistical properties, while data within a section are assumed to be homogeneous in certain ways.

As an example, a FITS[2] file consists of several header/data units (HDUs). `slim` should treat each header as one section of data and each binary table extension as another. The raw data as produced by the University of British Columbia's "Multichannel Electronics" (MCEs) consist of a single section if they are not too long.

Split section It is often necessary to split into several actual sections a run of data that is conceptually only a single section. For one thing, the implementation of `slim` requires entire sections to be held in memory at once, imposing a practical limit on the section

[2] *Flexible Image Transport System*, a format often used for astronomical data. We could make a similar example of NetCDF data files or other structured formats that consist mainly of data arrays.

size to some tens or hundreds of MB. (The current limit is 16MB.) It might also be true that the expected statistical properties of the data are not stationary over long times, which also indicates splitting a single section into several.

Frame Each section of data repeats its pattern of channels one or more times in a section of data. This repeated unit is called a *frame*. In the simplest data containing only one physical channel (e.g., in `dirfiles`), one frame is equivalent to and fills one section. For FITS data, a frame corresponds to one row of the binary extension. For the simplest multi-channel raw data, a frame is a single read of all the channels.

Channel The data to be compressed are produced by one or more instruments, physical sources of data. The values recorded by a single source are known collectively as a *channel*. A channel might correspond to a single optical detector or a thermometer, for example. Each frame consists of data from one or more channels, which may appear one or more times in a row before giving way to the next channel.

Repetitions How many consecutive instances of a channel's data appear in each frame. (Currently, `slim` has no way to handle a single channel appearing a two locations in a frame; such a situation would have to be treated as two distinct channels.)

Word size The data from a channel can be stored as words of 1, 2, or 4 bytes per datum. `slim` does not currently work for 8-byte data words.[3] Channels of different word size can be mixed together in a frame.

Type The data from each channel can be signed or unsigned integers. Floating-point numbers are handled as if they were signed integers.[4]

2.1 Example fatfile

To clarify the terms, let's look at an example fatfile. This file consists of two sections. The first section includes three channels A, B, and C. Let the ith data point from channel A be called a_i. Now suppose that each frame of data contains 4 repetitions of channel A, followed by one each of channels B and C. Let the first section contain 4 frames. The second section consists only of 8 data values from channel D. Then the data order will be (reading left to right, then top to bottom, as with English text):

$$\begin{array}{ll} \text{Section 1} & \left\{ \begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & b_0 & c_0 \\ a_4 & a_5 & a_6 & a_7 & b_1 & c_1 \\ a_8 & a_9 & a_{10} & a_{11} & b_2 & c_2 \\ a_{12} & a_{13} & a_{14} & a_{15} & b_3 & c_3 \end{array} \right. \\ \text{Section 2} & \left\{ \begin{array}{cccccccc} d_0 & d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 \end{array} \right. \end{array} \tag{1}$$

In this example, each line is one *frame* of data. The first four lines make up the first *section* of data, while the last line is the second section. Note that in the degenerate case of any section having only a single channel (like Section 2 in the example), it is a matter of convenience only

[3] For 8-byte floating point data, one could try tricks treating it as two 4-byte words.

[4] Currently, we have no plans to make `slim` target floating point numbers specifically, though suggestions are welcome. The author has a rough idea what he would do for floats, but he has no real incentive to follow through, as his own data sets are 99.5% integer data.

whether to say that the section is a single frame with several repetitions (as shown above) or several frames of one repetition each. Because `slim` works more efficiently with multiple repetitions than with multiple frames, the binary will always consider a single-channel section to be a single-frame section with multiple repetitions.

2.2 Non-science data

Some sections of a fatfile might consist of ASCII text or other non-science data. This could conceivably be true even for some channels within a section that is otherwise mostly science data, e.g. for a "channel" of fixed-length text comments or code characters that appears once per data frame. `slim` does not currently handle such data in any special way. For now, they must be treated as science channels and are "compressed" or not, depending on how their contents respond to the usual algorithms designed for science data.[5]

3 Specification of the `slim` compressed files

3.1 Bit ordering

The `slim` file can be considered as a stream of bits. The compression algorithm will fill these bits in order, N at a time, where N need not be a multiple of 8. However, we will implement `slim` on byte-oriented computers, which have a well-defined ordering of bytes in a stream but no intrinsic ordering of their bits (that is, is the lowest bit in a byte the "first" or last bit?). Therefore, for purposes of this program, we define the least significant bit of each byte to be "before" the others. For example, the byte `0xf0` = `11110000b` could indicate four 0-bits followed by four 1-bits, the unsigned number 240, or the signed number −16, depending on the context. In the first case, we would in this document write the stream from "first bit" to "last" as 0 0 0 0 1 1 1 1.

We also define byte 0 in a stream of bytes to be before byte 1. I picture all the bytes in a file or buffer being laid out from right-to-left, with byte 0 at the far right. Then the bits in that file can be treated as the bits of a single enormous number, with bits being written at (or read from) the least significant end first, that is, from right-to-left.

This byte-order choice is a good match to little-endian machines, because 16- and 32-bit numbers can be copied directly between the bit stream and the natural machine representation. Big-endian machines would have to reverse their byte order within multi-byte words when copying data literals between a raw and a slimfile. Both the ordering of bytes and of bits within a byte agree with the analogous conventions in the `gzip` compression system.

3.2 The slimfile and its file header

A complete slimfile consists of one overall file header and one or more data sections. The file header is detailed in Table 1. This brief header contains at minimum two bytes of file "magic", the raw file modification time, and an 8-bit field of flags describing the presence or absence of certain optional information in either the file header or the section headers. The file modification time appears to ensure proper handling of the timestamps on decompression.

[5] It is possible that a future release could permit long text sections to be compressed using another method, e.g. with `zlib`. But for the data files now anticipated by the author in his own work, the binary science data completely dominate the file contents.

Four flags (FLG.SIZE, FLG.NAME, FLG.XTRA, and FLG.TOC) say whether optional file header information is present: the uncompressed file size, the original file name (as a null-terminated string), an undefined "extra" header string of up to 65,536 bytes, and a section "table of contents." This latter consists of pointers in each section header to the start of the next data section, given as absolute file positions within the slimfile (suitable for a `fseek()` call with `whence=SEEK_SET`). One flag (FLG.ONECH) indicates whether all sections in the file are single-channel sections, and another (FLG.NOREP) indicates whether channels are always limited to only one repetition per data frame. If either of these flags are set, section headers are made shorter. It does not make sense to set both, and FLG.NOREP is ignored when FLG.ONECH is set. Finally, a flag (FLG.CRC) indicates whether a 32-bit CRC (a digest of the raw data) appears at the end of each data section.

After the flags, up to three optional items appear. First is the 32-bit size of the uncompressed file, in bytes (if FLG.SIZE was true). The standard does not specify what to do if the size exceeds 2^{32} bytes, unfortunately. Next the null-terminated filename string appears (if FLG.NAME was true). Then more items follow if FLG.XTRA was true: a 16-bit (unsigned integer) buffer size followed by the buffer itself. This specification does not address the contents of the extra buffer. Following these optional data is the header for the first section.

Test	Size (bits)	Data
	16	File magic: equals ASCII 'SL' = 0x4c53 (little endian)
	32	MTIME, last mod time of the raw file (seconds since epoch)
	8	FLG, a series of flags. In increasing bit significance they are:
		Val Name Signifies
		1x FLG.SIZE uncompressed file size present in header
		2x FLG.NAME original file name present in header
		4x FLG.XTRA extra data present in header
		8x FLG.TOC each section header has fwd ptr to next
		10x FLG.ONECH all sections have only one channel
		20x FLG.NOREP no channels repeat within any frame
		40x FLG.CRC CRC-32 appears at end of each secition
		80x FLG.X reserved for future use
FLG.SIZE	32	Uncompressed file size (bytes)
FLG.NAME	≥ 8	Original filename ('\0'-terminated string)
FLG.XTRA	16	Length of "extra bytes" that follow
FLG.XTRA	≥ 0	Up to 65,535 undefined extra bytes for any purpose

Table 1: Slim file header. All data in the file header are sized in integer bytes and are byte-aligned. The file header is followed by one or more data sections (see Section 3.3). In this and all other tables, the "Test" column means that a given element of the header appears if and only if the tested item is true. The author's usual usage is to set FLG.SIZE but not FLG.NAME or FLG.XTRA, which produces a fixed-length header of 11 bytes.

3.3 Sections

Sections are treated as completely distinct, even if consecutive sections have the same number and type of channels and were split merely to keep the section size below its limit. Thus, per-section information will repeat at the start of each section.

Test	Length (bits)	Data
	32	Size (bytes) of the *raw* data in this section
FLG.TOC	32	Absolute position (in bytes) of the next section header
NOT FLG.ONECH	24	N_c = # channels this section: $0 \leq N_c \leq 16,777,215$
		($N_c = 1$ implicitly if FLG.ONECH is set)
		(A number N_c of channel descriptions follow the header. See Table 3.)
		⋮
		(Section data block follows all N_c channel descriptions)
		⋮
FLG.CRC	32	CRC for the uncompressed section data
	4	End-of-section tag. Value indicates EOF or not:
		8x = Another section follows this one.
		ex = No sections follow (but there are "leftover bytes").
		fx = No sections follow (end of file marker).
		All other values for the tag are reserved.
Leftover bytes	3	N_b = # of leftover bytes: $1 \leq N_b \leq 7$
Leftover bytes	$8N_b$	The leftover bytes, as literal bytes, in order.

Table 2: Slim file section header, channel description block, and data block. The first bit is always byte-aligned (so that the reader can `fseek` to it). Up to seven bits after the end of the section have undefined values. Note in particular that the optional CRC need not be byte-aligned.

3.4 Frames

The section data block, which follows all N_c channel descriptions, consists of N_f successive slim data frames. The number N_f is not recorded in the file, and in fact there can be a non-integer number of frames. The section's raw data size (in the section header) governs how much slimmed data is to be read back. Each frame contains a compressed representation of the raw data words *in the same order* as the data appear in the fatfile. Thus the compressed data block consists of $N_r(0)$ consecutive values from channel 0, followed by $N_r(1)$ from channel 1, continuing through all N_c channels.

The representation of each data word depends on the algorithm being used to compress that data along with its channel-specific parameter set. See Section 4 for descriptions of these.

Test	Length (bits)	Data
$N_c > 1$ and not FLG.NOREP	24	$N_r(i)$ = Repetitions per frame: $0 \leq N_f \leq 16,777,215$
	1	DELTAS: Channel encodes data (0) or Δ(data) (1)
	5	Bit rotation count b.
	4	Algorithm code (see Table 4)
	4	Data type code (see Table 5)
	varies	Algorithm-specific data (see Section 4 and Table 6)

Table 3: Slim file channel description for channel i (N_c of these appear in succession after the section header and before the compressed data block). Note that if $N_c = 1$, then the $N_r(0)$ is computed from the section raw size (found in the section header);FLG.NOREP requires that $N_r(i) = 1$. In either of these cases, $N_r(i)$ would be redundant and is not recorded in the header.

Value	Algorithm name
0	Null encoder (copies words without change)
1	Reduced Binary encoder
2	Code A variant of Reduced Binary(*disabled*)
3	Code B variant of Reduced Binary(*disabled*)
4	Huffman coding (*disabled*)
5	Runlength coding
6	Constant Value coding

Table 4: Slim encoder algorithm codes. Disabled systems are discussed in Appendix B.

4 Compression algorithms

There are two compression algorithms available in `slim`. Most science data will use the reduced binary code (Section 4.2), which is the heart of the `slim` program. However, the runlength code (Section 4.3) is available for data which repeats bitwise-identical values for long periods.

4.1 The null algorithm

This is a dummy "compressor" that emits all raw data into the "compressed" output bit-for-bit. It is appropriate to use this on floating point types or any other types not yet instrumented with their own real compressor. This is probably also the right choice for short sections of ASCII data embedded in a science data file, e.g. a FITS header unit.

The null compressor does not have any algorithm-specific data.

4.2 Reduced binary codes

The idea of the reduced binary code is to identify the smallest possible contiguous segment of the 2^w possible integer values that contains substantially all of the values that appear in the channel (where w is the word size, in bits). This range is expanded to the next larger power of 2, 2^R, and the data are encoded as an R-bit number using the *reduced binary coding of order R* (Section A.2). This technique is valuable precisely because it is expected that $R < w$ for most

Value	Bits	Data type
0	—	*reserved*
1	32	unsigned int
2	32	signed int
3	16	unsigned short int
4	16	signed short int
5	32	IEEE 754 float
6	64	IEEE 754 double-precision float
7	8	unsigned char
8	8	signed char
9–15	—	*reserved*

Table 5: Slim encoder data type codes. (As of this writing, only the integer types are implemented. The 32-bit float is handled as a 32-bit signed integer. The 64-bit float is handled by the null encoder, i.e. its data are not affected by slimming.)

science data sets. One of the 2^R values is reserved to signal "overflow", i.e. a number that falls outside the allowed range. When overflow is signaled, the w bits following the overflow code hold the actual data.

For each reduced binary-coded channel, the basic parameters are R, the number of bits to use; and p, the channel "pedestal." The pedestal is the smallest data value that can be encoded without overflow. The pedestal need not be the actual minimum of the data, nor would it necessarily be optimal for the offset to equal the minimum. From these parameters, we derive the maximum nominal value, q:

$$q \quad = \quad p + 2^R - 2 \tag{2}$$

When $R = 1$-bit encoding is chosen, the maximum equals the minimum, and only the pedestal p fits the nominal range.

The compression algorithm for data value d is:

1. If $p \leq d \leq q$, then write $d' \equiv d - p$ as an R-bit number. You are done with this value.
2. Otherwise, write the special overflow code $-1 \equiv (2^R - 1)$ as an R-bit number. These numbers are equivalent in mod-2^R arithmetic, and have all R of their bits set.
3. After the overflow code, write the original value d (*not* the pedestal-subtracted value d') as a full w (8, 16, or 32)-bit number, matching the raw data word size.[6]

To decompress:

1. Read an R-bit number d'.
2. If $d' = 2^R - 1$ (the overflow code), discard d' and read d as a w-bit number.
3. Otherwise, compute and use $d = d' + p$.

[6]Since we are writing a full-size number anyway, it makes sense to write the one, d, which minimizes the number of transformations required at the decompression stage.

Length (bits)	Data
w	p, the pedestal
5	$R-1$, where $1 \le R \le 32$

Table 6: Reduced Binary code parameters, which appear in the channel description. R is the number of bits used to encode the nominal range of numbers; w is the raw word size (in bits).

See Appendix A.2 for more details on writing binary numbers in R bits, including the care that must be taken in reading in a negative signed number.

Note that there are various ways of choosing the parameters p and R for a data channel. In the first `slim` design, we took R to be the smallest value that allowed encoding of all the sample data. We next took p to be the number such that the midpoint of (p, q) matched the midpoint of the maximum and minimum sampled data values. This method can lead to setting R too high, as the choice of R is driven by potentially very rare extreme values of the data.

The current system of choosing R and p was for a time known as the Code A variation[7] This system instead starts by computing μ, the mean of the training data. Then `slim` tries all possible values of R and chooses the one that minimizes the total encoded length of the training set. For each trial value of R, the pedestal is set to $p = \mu - 2^{R-1}$, thereby centering the sample data mean μ in the middle of the nominal range. We found that this method for picking R and p improves the compression ratio of ACT bolometer and housekeeping data by approximately 2% without any detectable loss of compression speed. Decompression does not depend on how the parameters are chosen.

All arithmetic on d is done modulo 2^n where n is the raw wordsize. The only reason `slim` needs to know whether an integer is signed or unsigned is to know where the "branch cut" in the number line lies. That is, when testing a channel's data to find the range and thus the values of R and p to use, should it consider -1 and 0 to be neighbors on the number line (for signed integers) or maximally far apart (for unsigned integers)?

4.3 Runlength coding

Runlength codes are appropriate when a data value is constant over long periods of time and changes infrequently. Compression is achieved by storing the new data value along with the number of consecutive occurrences of that value. Whereas the reduced binary code requires at least 1 bit per data value, the runlength code can compress gigabytes of strictly constant data into a few bytes. However, the worst-case scenario, if the runlength coder is applied to random w-bit data, the "compressed" data will be nearly a factor of two times larger. When combined with the use of successive differences (deltas), runlength codes also work well with channels representing clocks and counters.

The compressed data are stored as the raw data value d (no pedestals are subtracted) and then the number of repeats m, both encoded by the modified exponential-Golomb code of order 1 (Section A.4) This is a code that favors shorter values with shorter codes, though it can be as long as 63 bits for the largest 50% of all 32-bit values. If the choice of encoding for d or m matters in some application, then Runlength coding is probably the wrong choice!

Note that some implementations (including the current version) will not be able to count repeats beyond a frame boundary for data having more than one frame per section. Although

[7] I mention this in case you come upon it in the source code.

this deficiency hurts compression performance, it does not affect the present discussion of file formats. On the other hand, if some future implementation ever manages to write a large m exceeding the number of repetitions of a channel in one frame, then such usage *is* consistent with the `slim` file format.

The Runlength code does not have any parameters stored in the channel description part of the section header. Its operation does depend on whether the channel contains signed or unsigned data, as the modified exponential-Golomb code is implemented differently for signed and unsigned values.

4.4 Constant Value coding

Some channels' data is strictly constant. These data are encoded by the Constant Value encoder. It contains a single parameter in the channel description, which is the constant value itself and which has the same length as the channel's data word. The section data block contains no data at all from the Constant Value channel. Although this is very similar to Reduced Binary coding in the limit of $R = 0$, we find it more efficient to treat it as a separate case.

5 Bit rotation

Bit rotation is a feature added when we noticed that some data channels have raw data like the following series[8] of 32-bit data:

```
87f71300 87f71800 87f71600 87f71200 87f71300 87f71600 87f71200 87f70d00
87f71200 87f71800 87f71a00 87f71a00 87f71a00 87f71800 87f71100 87f70d00
```

The bits that vary from one sample to another are not the lowest few, but the bits 8–12; the lowest eight bits are always zero. In this channel, the cause is related to the digital filter that processes the data internally as 32-bit floats (which have eight bits reserved for the exponent) before converting them back to integers. But similar data with fixed low-order bits can also result whenever a data channel packs status information into the low-order bits.

The `slim` method to handle such values is designed to be as simple as possible, without presuming that the static bits are all zero (or any other specific value). When asked to check for static low bits, `slim` will find the largest number b of strictly constant low-order bits. When $b > 0$, `slim` replaces the raw value with a cyclic permutation of its bits, right-shifted by b. This brings the static bits to the top of the value, at which point the reduced binary encoding system can be applied effectively to the bit-permuted value. On decoding, the last step is to apply the cyclic permutation in the opposite direction. Because all data in a channel share the same number b, it needs to be recorded only once in the channel description (Table 3).

A Systems for representing single values

The coding systems described here represent single values in more than one way, depending on the anticipated probability distribution of the values in practice. We prescribe in detail the most common. Some vocabulary:

[8] From a cryogenic diode thermometer.

symbols The data values being transformed, that is, the raw data. In the present application, the symbols are the possible values of the data streams (usually the set of 32-bit signed or unsigned integers, though 8- or 16-bit values are also possible).

codes The transformed data, from which the original symbols can be recreated. Good compression means making the average code as short as possible. Codes are an integer number of bits long.[9]

coding system An algorithm for interconverting between unique symbols and unique codes. The goal in compression is to use the shortest codes for the most common symbols. The ideal code lengths are specified by the Shannon entropy theorem.

universal code A coding system that can in principle describe all numbers of any magnitude. The opposite, non-universal codes, have a limited domain to which they can be applied without "collisions" between the codes for more than one value. For our purposes, a code that can encode all possible 32-bit values is as good as universal.

fixed-length code A coding system in which every code is of the same length (in bits). Such codes are inefficient if some words in are much more likely to occur than others.

prefix-free code A coding system with variable-length codes having the "prefix property," meaning that no valid code is equivalent to the leading bits of any other valid, longer code. With this property, it is possible to discern the code's length from the code itself. Telephone dialing is an example, as I can dial locally (924-1234), long distance (1-979-297-9700), or internationally (011-56-9-7746-5001). These codes are 7, 11, and 14 digits long, and the leading digits determine how many more digits are required to complete the code.

A.1 Signed and unsigned integers

All additions and subtractions in `slim` are done in modulo-2^n arithmetic for n-bit integers. This is true whether the data are regarded as signed or unsigned integers. Throughout, we assume that negative numbers are represented in twos-complement notation (which is employed on all workstations in common use, to the best of our knowledge).

A.2 Reduced binary coding

Reduced binary coding is the workhorse of the `slim` program. It is exceedingly fast, in situations where the number is known to fit in the domain.

In *reduced binary coding of order k*, numbers are represented by their k least significant bits only. Obviously, this is not a universal code, because only values in the range $[0, 2^k)$ can be represented uniquely. The reduced binary code is of fixed length. For example, the order-4 reduced binary code has a domain of $[0, 15]$ and in pseudocode it is encoded:

```
k = 4
mask = (1 << k) - 1;   // = ...0000 0000 1111
code = value & mask;
```

[9] Arithmetic coding puts a data value into a non-integer number of bits in the output, and so the notion of a single code for each raw symbol would not apply. `slim` does not support arithmetic coding.

```
writebits(code, k);
```

To decode the order-4 reduced binary code:

```
k = 4;
value = code = readbits(k);
```

In every case, the size of the code is $s = k$.

In `slim`, we extend this system slightly to make the coding universal. The code -1 (or equivalently, $2^k - 1$) is reserved to indicate an overflow, either a negative number or one greater than or equal to $2^k - 1$. In case of an overflow, the full overflowing value is written after the overflow signal.

`slim` avoids handling signed integers (and the attendant problems with properly setting the highest bits on negative numbers) by taking as a pedestal value a number at least as negative as the most negative value to be encoded (without overflow). All data are read back in as unsigned integers and brought to the proper range through the addition of the pedestal.

A.3 Unary coding

Unary coding is a universal, prefix-free code for the whole numbers. `slim` uses unary coding as the first component in exponential-Golomb coding (Section A.4) to represent the number of bits in the main component. Unary coding represents the whole number n as a series of n 1s followed by a single 0. Thus

n	Code
0	0
1	1 0
2	1 1 0
3	1 1 1 0

The size $s(n)$ of every code is $s = 1 + n$.

This code is optimal for any geometric distribution falling rapidly enough, meaning that each symbol is a factor of at least $\phi = 1.618...$ less probable than the one before.

For comparison to other uses of unary coding, note that some sources reverse the use of the 0 and 1 bits as a matter of convention. Also, some sources omit zero from the encodable set, choosing to code n as $n - 1$ 1s and a 0, which reduces the length of each code by one.

A.4 Exponential-Golomb coding

Exponential-Golomb coding of order k is used in the runlength algorithm to store both the raw value and its repetition count. It is similar to reduced binary coding, except that a "size prefix" is prepended to specify the length of the data value rather than having it agreed to in advance. Because this prefix is a universal code, and because it announces the size of the rest of the code, the system as a whole is also a universal code.

The idea is to write the symbol in a reduced binary code of the smallest possible number of bits b, prefixing it by the length minus the order of the code (that is, the value of $b - k$), represented by a unary code. The code's order k is chosen to be the minimum value expected for b in "typical" use. Thus the prefix is the unary code for $b - k$, and b is never allowed to be less than k. Note that when $b > k$, the highest bit is guaranteed to be one. In that case, therefore, it would be redundant to write the highest bit. Only the $b-1$ lowest bits are written. In pseudocode:

```
// Encoding
b = k;
while (2^b <= value)    b++;
write_unary(b-k);
if (b>k)
  writebits(value, b-1);
else
  writebits(value, k);

// Decoding
b = k + read_unary();
if (b>k)
  value = readbits(b-1) | ( 1 << (b-1) );
else
  value = readbits(k);
```

Thus

n	Code (order 1)	Code (order 2)	Code (order 3)
0	0, 0	0, 00	0, 000
1	0, 1	0, 01	0, 001
2	1 0, 0	0, 10	0, 010
3	1 0, 1	0, 11	0, 011
4	1 1 0, 00	1 0, 00	0, 100
5	1 1 0, 01	1 0, 01	0, 101
6	1 1 0, 10	1 0, 10	0, 110
7	1 1 0, 11	1 0, 11	0, 111
8	1 1 1 0, 000	1 1 0, 000	1 0, 000
9	1 1 1 0, 001	1 1 0, 001	1 0, 001

The code size $s(n)$ for unsigned data n is

$$s(n) = \begin{cases} 1+k, & n < 2^k \\ 2+k, & 2^k \le n < 2^{k+1} \\ 2+2\lfloor \log_2 n \rfloor - k, & 2^{k+1} \le n \end{cases}$$

B Code ideas tried but no longer used

Some ideas for improving on the reduced binary coding scheme have been implemented in `slim` but were later rejected. They have been removed from the program to simplify the C++ source and to make the executable smaller. Code B and the Huffman code were written into `slim` and then removed; Code C was never written for `slim`. We discuss them here in brief just to point out for the ambitious data compressors of the future what has been tried and found unhelpful (or rather, not helpful enough to be worth the trouble).

B.1 Code B variant of the reduced binary code

Code B differs from the Reduced Binary code in its handling of possible overflows outside the nominal range $[p, p+2^R-2]$. Overflows are written in a 3-step code, which is more complicated

but generally shorter than simply emitting the full data value d as a 32-bit number. The idea of Code B is that most overflow values will be close to the normal range of values, rather than being spread out over the full 32-bit range. For example, when using a reduced binary code of order 16, the typical overflow will be a 17- or 18-bit number. Better, then, to try not to waste 32 bits on it.

The second concept is that overflows should occur a few percent of the time—much more often than most other single data values—and so the symbol for an overflow should be shorter than the other codes. We let s be the size (in bits) of the overflow code. We anticipate $s \sim 4$ to 6, but for any channel it is necessary that $s \leq R$. This means that on decoding, the program must first read s bits and compare against the overflow code. If the data do not match, then the remaining $R - s$ bits are read next. So normal processing of data unfortunately requires 2 reads from the bitstream, which slows down decoding.

In practice, we find that the Code B variant improves on the compression ratio by only about 1%, but it slows down compression and decompression by 25% to 50%. (The slowdown is more serious for very large files, where the operations are limited by CPU; slimming a large number of small files tends to have large overhead in the operating system and filesystem interactions.)

B.2 Code C variant of the reduced binary code

Code C was designed on paper but never coded as `slim` objects nor used in the `slim` program. The idea was to improve on the reduced binary coding (or Code B) by using truncated binary encoding, which generalizes the idea by permitting the nominal range not to be a power of 2. Calculations of compressed data sizes on real science data sets showed that Code C would improve on Code B by only $\sim 0.1\%$ on average. Therefore, it was never tried in `slim`.

B.3 Huffman coding

Huffman coding is a near-optimal system for encoding a data stream if the exact distribution of symbols is known in advance. If we further require that each symbol be represented by a fixed code with an integer number of bits (i.e. each output bit is determined by only one input symbol), then Huffman coding is in fact optimal. (Note that range coding and arithmetic coding mix symbols into codes in such as way as to assign codes what amounts to fractional bits, thereby improving on Huffman codes in size, if not in speed.) Even if we only sample the data set to predict the full distribution, as with `slim`, we find that Huffman coding provides the best compression of all systems described in this document, roughly 3% to 5% better than the reduced binary code.

Unfortunately, Huffman coding is also the slowest of the systems, compressing data at only 50% of the speed of the reduced binary code and decompressing at only 40%. It is possible that we could improve these numbers somewhat, but the compression gains would probably be too modest to justify. One difficulty is that Huffman codes are not of a fixed length. Although they make up a prefix-free code, one must in general read the bits one at a time to determine whether more bits are needed. This feature slows down the decoding process considerably.