



WIKIPEDIA
The Free Encyclopedia

[Main page](#)
[Contents](#)
[Featured content](#)
[Current events](#)
[Random article](#)
[Donate to Wikipedia](#)
[Wikipedia store](#)

Interaction

[Help](#)
[About Wikipedia](#)
[Community portal](#)
[Recent changes](#)
[Contact page](#)

Tools

[What links here](#)
[Related changes](#)
[Upload file](#)
[Special pages](#)
[Permanent link](#)
[Page information](#)
[Wikidata item](#)
[Cite this page](#)

Print/export

[Create a book](#)
[Download as PDF](#)
[Printable version](#)

Languages

[Čeština](#)
[Deutsch](#)
[Français](#)
[한국어](#)
[Italiano](#)
[Nederlands](#)
[日本語](#)
[Polski](#)
[Português](#)
[Русский](#)
[Українська](#)
[中文](#)

[Edit links](#)

[Create account](#) [Log in](#)

Article [Talk](#)

[Read](#) [Edit](#) [View history](#)

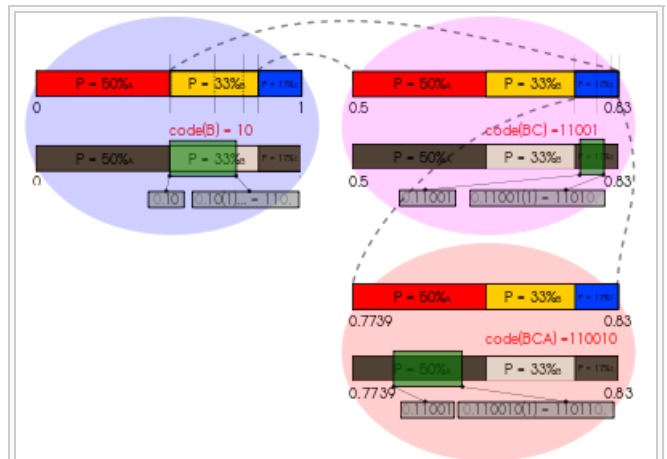
Arithmetic coding

From Wikipedia, the free encyclopedia

Arithmetic coding is a form of [entropy encoding](#) used in [lossless data compression](#). Normally, a string of characters such as the words "hello there" is represented using a fixed number of bits per character, as in the [ASCII](#) code. When a string is converted to arithmetic encoding, frequently used characters will be stored with fewer bits and not-so-frequently occurring characters will be stored with more bits, resulting in fewer bits used in total. Arithmetic coding differs from other forms of entropy encoding, such as [Huffman coding](#), in that rather than separating the input into component symbols and replacing each with a code, arithmetic coding encodes the entire message into a single number, a fraction *n* where ($0.0 \leq n < 1.0$).

Contents

- 1 Implementation details and examples
 - 1.1 Equal probabilities
 - 1.2 Defining a model
 - 1.3 Encoding and decoding: overview
 - 1.4 Encoding and decoding: example
 - 1.5 Sources of inefficiency
- 2 Adaptive arithmetic coding
- 3 Precision and renormalization
- 4 Arithmetic coding as a generalized change of radix
 - 4.1 Theoretical limit of compressed message
- 5 p-adic interpretation of arithmetic coding algorithm
- 6 Connections with other compression methods
 - 6.1 Huffman coding
- 7 US patents
- 8 Benchmarks and other technical characteristics
- 9 See also
- 10 Notes
- 11 References
- 12 External links



An arithmetic coding example assuming a fixed probability distribution of three Symbols "A", "B", and "C". Probability of "A" is 50%, probability of "B" is 33% and probability of "C" is 17%. Furthermore we assume that the recursion depth is known in each step. In step one we code "B" which is inside the interval (0.5, 0.83): The binary number "0.10x" is the shortest code that represents an Interval that is entirely inside [0.5, 0.83]. "x" means an arbitrary bit sequence. There are two extreme cases: the smallest x stands for an infinite number of zeros which represents the left side of the represented interval. Then the left side of the interval is $\text{dec}(0.10) = 0.5$. The greatest x stands for an infinite number of ones which gives a number that converges towards $\text{dec}(0.11) = 0.75$. Therefore "0.10x" represents the interval [0.5, 0.75] which is inside [0.5, 0.83]. Now we can leave out the "0." part since all intervals begin with "0." and we can ignore the "x" part because no matter what bit-sequence it represents, we will stay inside [0.5, 0.75].

Implementation details and examples

Equal probabilities

In the simplest case, the probability of each symbol occurring is equal. For example, consider a set of three symbols, A, B, and C, each equally likely to occur. Simple [block encoding](#) would require 2 bits per symbol, which is wasteful: one of the bit variations is never used. That is to say, A=00, B=01, and C=10, but 11 is unused.

A more efficient solution is to represent a sequence of these three symbols as a rational number in base 3 where each digit represents a symbol. For example, the sequence "ABBCAB" could become 0.011201_3 (in arithmetic coding the numbers are between 0 and 1). The next step is to encode this [ternary](#) number using a fixed-point binary number of sufficient precision to recover it, such as 0.0010110010_2 — this is only 10 bits; 2 bits are saved in comparison with naïve block encoding. This is feasible for long sequences because there are efficient, in-place algorithms for converting the base of arbitrarily precise numbers.

To decode the value, knowing the original string had length 6, one can simply convert back to base 3, round to 6 digits, and recover the string.

Defining a model

In general, arithmetic coders can produce near-optimal output for any given set of symbols and probabilities (the optimal value is $-\log_2 P$ bits for each symbol of probability P , see [source coding theorem](#)). Compression algorithms that use arithmetic coding start by determining a [model](#) of the data – basically a prediction of what patterns will be found in the symbols of the message. The more accurate this prediction is, the closer to optimal the output will be.

Example: a simple, static model for describing the output of a particular monitoring instrument over time might be:

- 60% chance of symbol NEUTRAL
- 20% chance of symbol POSITIVE
- 10% chance of symbol NEGATIVE
- 10% chance of symbol END-OF-DATA. (*The presence of this symbol means that the stream will be 'internally terminated', as is fairly common in data compression; when this symbol appears in the data stream, the decoder will know that the entire stream has been decoded.*)

Models can also handle alphabets other than the simple four-symbol set chosen for this example. More sophisticated models are also possible: *higher-order* modelling changes its estimation of the current probability of a symbol based on the symbols that precede it (the *context*), so that in a model for English text, for example, the percentage chance of "u" would be much higher when it follows a "Q" or a "q". Models can even be [adaptive](#), so that they continually change their prediction of the data based on what the stream actually contains. The decoder must have the same model as the encoder.

Encoding and decoding: overview [\[edit\]](#)

In general, each step of the encoding process, except for the very last, is the same; the encoder has basically just three pieces of data to consider:

- The next symbol that needs to be encoded
- The current [interval](#) (at the very start of the encoding process, the interval is set to $[0,1]$, but that will change)
- The probabilities the model assigns to each of the various symbols that are possible at this stage (as mentioned earlier, higher-order or adaptive models mean that these probabilities are not necessarily the same in each step.)

The encoder divides the current interval into sub-intervals, each representing a fraction of the current interval proportional to the probability of that symbol in the current context. Whichever interval corresponds to the actual symbol that is next to be encoded becomes the interval used in the next step.

Example: for the four-symbol model above:

- the interval for NEUTRAL would be $[0, 0.6]$
- the interval for POSITIVE would be $[0.6, 0.8]$
- the interval for NEGATIVE would be $[0.8, 0.9]$
- the interval for END-OF-DATA would be $[0.9, 1]$.

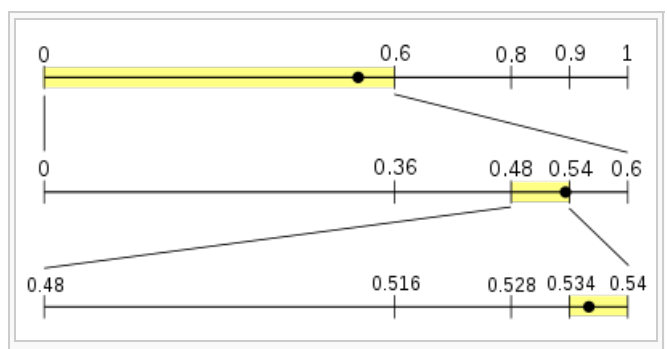
When all symbols have been encoded, the resulting interval unambiguously identifies the sequence of symbols that produced it. Anyone who has the same final interval and model that is being used can reconstruct the symbol sequence that must have entered the encoder to result in that final interval.

It is not necessary to transmit the final interval, however; it is only necessary to transmit *one fraction* that lies within that interval. In particular, it is only necessary to transmit enough digits (in whatever base) of the fraction so that all fractions that begin with those digits fall into the final interval; this will guarantee that the resulting code is a [prefix code](#).

Encoding and decoding: example [\[edit\]](#)

Consider the process for decoding a message encoded with the given four-symbol model. The message is encoded in the fraction 0.538 (using decimal for clarity, instead of binary; also assuming that there are only as many digits as needed to decode the message.)

The process starts with the same interval used by the encoder: $[0,1]$, and using the



same model, dividing it into the same four sub-intervals that the encoder must have. The fraction 0.538 falls into the sub-interval for NEUTRAL, [0, 0.6); this indicates that the first symbol the encoder read must have been NEUTRAL, so this is the first symbol of the message.

Next divide the interval [0, 0.6) into sub-intervals:

- the interval for NEUTRAL would be [0, 0.36), 60% of [0, 0.6).
- the interval for POSITIVE would be [0.36, 0.48), 20% of [0, 0.6).
- the interval for NEGATIVE would be [0.48, 0.54), 10% of [0, 0.6).
- the interval for END-OF-DATA would be [0.54, 0.6), 10% of [0, 0.6).

Since .538 is within the interval [0.48, 0.54), the second symbol of the message must have been NEGATIVE.

Again divide our current interval into sub-intervals:

- the interval for NEUTRAL would be [0.48, 0.516).
- the interval for POSITIVE would be [0.516, 0.528).
- the interval for NEGATIVE would be [0.528, 0.534).
- the interval for END-OF-DATA would be [0.534, 0.540).

Now 0.538 falls within the interval of the END-OF-DATA symbol; therefore, this must be the next symbol. Since it is also the internal termination symbol, it means the decoding is complete. If the stream is not internally terminated, there needs to be some other way to indicate where the stream stops. Otherwise, the decoding process could continue forever, mistakenly reading more symbols from the fraction than were in fact encoded into it.

Sources of inefficiency [\[edit\]](#)

The message 0.538 in the previous example could have been encoded by the equally short fractions 0.534, 0.535, 0.536, 0.537 or 0.539. This suggests that the use of decimal instead of binary introduced some inefficiency. This is correct; the information content of a three-digit decimal is approximately 9.966 bits; the same message could have been encoded in the binary fraction 0.10001010 (equivalent to 0.5390625 decimal) at a cost of only 8 bits. (The final zero must be specified in the binary fraction, or else the message would be ambiguous without external information such as compressed stream size.)

This 8 bit output is larger than the information content, or [entropy](#) of the message, which is

$$\sum -\log_2(p_i) = -\log_2(0.6) - \log_2(0.1) - \log_2(0.1) = 7.381 \text{ bits}$$

But an integer number of bits must be used, so an encoder for this message would use at least 8 bits, on average, which is achieved by the coding method, resulting in a message 8.4% larger than the minimum. This inefficiency of at most 1 bit becomes less significant as the message size grows.

Moreover, the claimed symbol probabilities were [0.6, 0.2, 0.1, 0.1], but the actual frequencies in this example are [0.33, 0, 0.33, 0.33]. If the intervals are readjusted for these frequencies, the entropy of the message would be 4.755 bits and the same NEUTRAL NEGATIVE ENDOFDATA message could be encoded as intervals [0, 1/3); [1/9, 2/9); [5/27, 6/27); and a binary interval of [0.00101111011, 0.00111000111). This is also an example of how statistical coding methods like arithmetic encoding can produce an output message that is larger than the input message, especially if the probability model is off.

Adaptive arithmetic coding [\[edit\]](#)

One advantage of arithmetic coding over other similar methods of data compression is the convenience of adaptation. *Adaptation* is the changing of the frequency (or probability) tables while processing the data. The decoded data matches the original data as long as the frequency table in decoding is replaced in the same way and in the same step as in encoding. The synchronization is, usually, based on a combination of symbols occurring during the encoding and decoding process.

Precision and renormalization [\[edit\]](#)

The above explanations of arithmetic coding contain some simplification. In particular, they are written as if the encoder first calculated the fractions representing the endpoints of the interval in full, using infinite [precision](#), and only converted the fraction to its final form at the end of encoding. Rather than try to simulate infinite precision, most arithmetic coders instead operate at a fixed limit of precision which they know the decoder will

A diagram showing decoding of 0.538 (the circular point) in the example model. The region is divided into subregions proportional to symbol frequencies, then the subregion containing the point is successively subdivided in the same way.



be able to match, and round the calculated fractions to their nearest equivalents at that precision. An example shows how this would work if the model called for the interval $[0,1)$ to be divided into thirds, and this was approximated with 8 bit precision. Note that since now the precision is known, so are the binary ranges we'll be able to use.

Symbol	Probability (expressed as fraction)	Interval reduced to eight-bit precision (as fractions)	Interval reduced to eight-bit precision (in binary)	Range in binary
A	1/3	$[0, 85/256)$	$[0.00000000, 0.01010101)$	00000000 – 01010100
B	1/3	$[85/256, 171/256)$	$[0.01010101, 0.10101011)$	01010101 – 10101010
C	1/3	$[171/256, 1)$	$[0.10101011, 1.00000000)$	10101011 – 11111111

A process called *renormalization* keeps the finite precision from becoming a limit on the total number of symbols that can be encoded. Whenever the range is reduced to the point where all values in the range share certain beginning digits, those digits are sent to the output. For however many digits of precision the computer *can* handle, it is now handling fewer than that, so the existing digits are shifted left, and at the right, new digits are added to expand the range as widely as possible. Note that this result occurs in two of the three cases from our previous example.

Symbol	Probability	Range	Digits that can be sent to output	Range after renormalization
A	1/3	00000000 – 01010100	0	00000000 – 10101001
B	1/3	01010101 – 10101010	None	01010101 – 10101010
C	1/3	10101011 – 11111111	1	01010110 – 11111111

Arithmetic coding as a generalized change of radix [\[edit\]](#)

Recall that in the case where the symbols had equal probabilities, arithmetic coding could be implemented by a simple change of base, or radix. In general, arithmetic (and range) coding may be interpreted as a *generalized* change of [radix](#). For example, we may look at any sequence of symbols:

DABDDDB

as a number in a certain base presuming that the involved symbols form an ordered set and each symbol in the ordered set denotes a sequential integer $A = 0$, $B = 1$, $C = 2$, $D = 3$, and so on. This results in the following frequencies and cumulative frequencies:

Symbol	Frequency of occurrence	Cumulative frequency
A	1	0
B	2	1
D	3	3

The *cumulative frequency* is the total of all frequencies below it in a frequency distribution (a running total of frequencies).

In a positional [numeral system](#) the radix, or base, is numerically equal to a number of different symbols used to express the number. For example, in the decimal system the number of symbols is 10, namely 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The radix is used to express any finite integer in a presumed multiplier in polynomial form. For example, the number 457 is actually $4 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$, where base 10 is presumed but not shown explicitly.

Initially, we will convert DABDDDB into a base-6 numeral, because 6 is the length of the string. The string is first mapped into the digit string 301331, which then maps to an integer by the polynomial:

$$6^5 \times 3 + 6^4 \times 0 + 6^3 \times 1 + 6^2 \times 3 + 6^1 \times 3 + 6^0 \times 1 = 23671$$

The result 23671 has a length of 15 bits, which is not very close to the theoretical limit (the [entropy](#) of the

message), which is approximately 9 bits.

To encode a message with a length closer to the theoretical limit imposed by information theory we need to slightly generalize the classic formula for changing the radix. We will compute lower and upper bounds L and U and choose a number between them. For the computation of L we multiply each term in the above expression by the product of the frequencies of all previously occurred symbols:

$$\begin{aligned} L &= (6^5 \times 3) + \\ &\quad 3 \times (6^4 \times 0) + \\ &\quad (3 \times 1) \times (6^3 \times 1) + \\ &\quad (3 \times 1 \times 2) \times (6^2 \times 3) + \\ &\quad (3 \times 1 \times 2 \times 3) \times (6^1 \times 3) + \\ &\quad (3 \times 1 \times 2 \times 3 \times 3) \times (6^0 \times 1) \\ &= 25002 \end{aligned}$$

The difference between this polynomial and the polynomial above is that each term is multiplied by the product of the frequencies of all previously occurring symbols. More generally, L may be computed as:

$$L = \sum_{i=1}^n n^{n-i} C_i \prod_{k=1}^{i-1} f_k$$

where C_i are the cumulative frequencies and f_k are the frequencies of occurrences. Indexes denote the position of the symbol in a message. In the special case where all frequencies f_k are 1, this is the change-of-base formula.

The upper bound U will be L plus the product of all frequencies; in this case $U = L + (3 \times 1 \times 2 \times 3 \times 3 \times 2) = 25002 + 108 = 25110$. In general, U is given by:

$$U = L + \prod_{k=1}^n f_k$$

Now we can choose any number from the interval $[L, U)$ to represent the message; one convenient choice is the value with the longest possible trail of zeroes, 25100, since it allows us to achieve compression by representing the result as 251×10^2 . The zeroes can also be truncated, giving 251, if the length of the message is stored separately. Longer messages will tend to have longer trails of zeroes.

To decode the integer 25100, the polynomial computation can be reversed as shown in the table below. At each stage the current symbol is identified, then the corresponding term is subtracted from the result.

Remainder	Identification	Identified symbol	Corrected remainder
25100	$25100 / 6^5 = 3$	D	$(25100 - 6^5 \times 3) / 3 = 590$
590	$590 / 6^4 = 0$	A	$(590 - 6^4 \times 0) / 1 = 590$
590	$590 / 6^3 = 2$	B	$(590 - 6^3 \times 1) / 2 = 187$
187	$187 / 6^2 = 5$	D	$(187 - 6^2 \times 3) / 3 = 26$
26	$26 / 6^1 = 4$	D	$(26 - 6^1 \times 3) / 3 = 2$
2	$2 / 6^0 = 2$	B	

During decoding we take the floor after dividing by the corresponding power of 6. The result is then matched against the cumulative intervals and the appropriate symbol is selected from look up table. When the symbol is identified the result is corrected. The process is continued for the known length of the message or while the remaining result is positive. The only difference compared to the classical change-of-base is that there may be a range of values associated with each symbol. In this example, A is always 0, B is either 1 or 2, and D is any of 3, 4, 5. This is in exact accordance with our intervals that are determined by the frequencies. When all intervals are equal to 1 we have a special case of the classic base change.

Theoretical limit of compressed message [\[edit\]](#)

The lower bound L never exceeds n^n , where n is the size of the message, and so can be represented in $\log_2(n^n) = n \log_2(n)$ bits. After the computation of the upper bound U and the reduction of the message by selecting a number from the interval $[L, U)$ with the longest trail of zeros we can presume that this length can be reduced by $\log_2\left(\prod_{k=1}^n f_k\right)$ bits. Since each frequency in a product occurs exactly same number of times as the value of this frequency, we can use the size of the alphabet A for the computation of the product

$$\prod_{k=1}^n f_k = \prod_{k=1}^A f_k^{f_k}$$

Applying \log_2 for the estimated number of bits in the message, the final message (not counting a logarithmic overhead for the message length and frequency tables) will match the number of bits given by [entropy](#), which for long messages is very close to optimal:

$$n \log_2(n) - \sum_{i=1}^A f_i \log_2(f_i)$$

p-adic interpretation of arithmetic coding algorithm [\[edit\]](#)

Arithmetic coding being expressed in terms of real numbers looks very natural and is easy to understand. It is nothing but a sequence of semi intervals each lies inside the previous one. But here is a problem – one has to use infinite precision real numbers to implement this algorithm and there is no such a thing like effective infinite precision real arithmetic. This problem was always considered as a technical one. Solution is simple - just use integers instead. There is a canonical implementation, first written in C [Witten], which was later reproduced in other languages, but no analysis of what happens to the algorithm after moving it from the real numbers to integer numbers was published. In fact, the integer variant of the algorithm looks very artificial and contains some magic rules: E1, E2 and E3. Though these rules work quite well the question remains – do they have natural mathematical explanation?

The [p-adic numbers](#) provides clear interpretation of the algorithm. In fact, all the intermediate data and the result can be seen as p-adic integers with $p=2$. The modified algorithm operates on p-adic semi intervals in the same way, as the original works with real semi intervals. For example the magic rules E1, E2 mean that the current p-adic semi interval lies completely in a p-adic ball. In this case the p-adic ball can be pushed out and p-adic semi interval rescaled. From this point of view Huffman algorithm is just a specific variant of arithmetic coding when semi intervals are always p-adic balls.

The algorithm can be extended to arbitrary p . All E1, E2, and E3 rules work in this case too. More information on p-adic variant of arithmetic coding can be found in [Rodionov, Volkov 2007, 2010].

Connections with other compression methods [\[edit\]](#)

Huffman coding [\[edit\]](#)

Main article: [Huffman coding](#)

There is great similarity between arithmetic coding and Huffman coding – in fact, it has been shown that Huffman is just a specialized case of arithmetic coding – but because arithmetic coding translates the entire message into one number represented in [base \$b\$](#) , rather than translating each symbol of the message into a series of digits in base b , it will sometimes approach optimal [entropy encoding](#) much more closely than Huffman can.

In fact, a Huffman code corresponds closely to an arithmetic code where each of the frequencies is rounded to a nearby power of $\frac{1}{2}$ — for this reason, Huffman deals relatively poorly with distributions where symbols have frequencies far from a power of $\frac{1}{2}$, such as 0.75 or 0.375. This includes most distributions where there are either a small number of symbols (such as just the bits 0 and 1) or where one or two symbols dominate the rest.

For an alphabet {a, b, c} with equal probabilities of 1/3, Huffman coding may produce the following code:

- a → 0: 50%
- b → 10: 25%
- c → 11: 25%

This code has an expected $(2 + 2 + 1)/3 \approx 1.667$ bits per symbol for Huffman coding, an inefficiency of 5 percent compared to $\log_2 3 \approx 1.585$ bits per symbol for arithmetic coding.

For an alphabet {0, 1} with probabilities 0.625 and 0.375, Huffman encoding treats them as though they had 0.5 probability each, assigning 1 bit to each value, which does not achieve any compression over naive block encoding. Arithmetic coding approaches the optimal compression ratio of

$$1 - [-0.625 \log_2(0.625) + -0.375 \log_2(0.375)] \approx 4.6\%.$$

When the symbol 0 has a high probability of 0.95, the difference is much greater:

$$1 - [-0.95 \log_2(0.95) + -0.05 \log_2(0.05)] \approx 71.4\%.$$

One simple way to address this weakness is to concatenate symbols to form a new alphabet in which each

symbol represents a sequence of symbols in the original alphabet. In the above example, grouping sequences of three symbols before encoding would produce new "super-symbols" with the following frequencies:

- 000: 85.7%
- 001, 010, 100: 4.5% each
- 011, 101, 110: 0.24% each
- 111: 0.0125%

With this grouping, Huffman coding averages 1.3 bits for every three symbols, or 0.433 bits per symbol, compared with one bit per symbol in the original encoding.

US patents [[edit](#)]

A variety of specific techniques for arithmetic coding have historically been covered by US [patents](#), although various well-known methods have since passed into the public domain as the patents have expired. Techniques covered by patents may be essential for implementing the algorithms for arithmetic coding that are specified in some formal international standards. When this is the case, such patents are generally available for licensing under what is called "reasonable and non-discriminatory" ([RAND](#)) licensing terms (at least as a matter of standards-committee policy). In some well-known instances, (including some involving IBM patents that have since expired), such licenses were available for free, and in other instances, licensing fees have been required. The availability of licenses under RAND terms does not necessarily satisfy everyone who might want to use the technology, as what may seem "reasonable" for a company preparing a proprietary software product may seem much less reasonable for a [free software](#) or [open source](#) project.

At least one significant compression software program, [bzip2](#), deliberately discontinued the use of arithmetic coding in favor of Huffman coding due to the perceived patent situation at the time. Also, encoders and decoders of the [JPEG](#) file format, which has options for both Huffman encoding and arithmetic coding, typically only support the Huffman encoding option, which was originally because of patent concerns; the result is that nearly all JPEG images in use today use Huffman encoding^[1] although JPEG's arithmetic coding patents^[2] have expired due to the age of the JPEG standard (the design of which was approximately completed by 1990).^[3] There are some archivers like PackJPG, that can losslessly convert Huffman encoded files to ones with arithmetic coding (with custom file name .jpg), showing up to 25% size saving.

Some US patents relating to arithmetic coding are listed below.

- [U.S. Patent 4,122,440](#) [[↗](#)] — (IBM) Filed 4 March 1977, Granted 24 October 1978 (Now expired)
- [U.S. Patent 4,286,256](#) [[↗](#)] — (IBM) Granted 25 August 1981 (Now expired)
- [U.S. Patent 4,467,317](#) [[↗](#)] — (IBM) Granted 21 August 1984 (Now expired)
- [U.S. Patent 4,652,856](#) [[↗](#)] — (IBM) Granted 4 February 1986 (Now expired)
- [U.S. Patent 4,891,643](#) [[↗](#)] — (IBM) Filed 15 September 1986, granted 2 January 1990 (Now expired)
- [U.S. Patent 4,905,297](#) [[↗](#)] — (IBM) Filed 18 November 1988, granted 27 February 1990 (Now expired)
- [U.S. Patent 4,933,883](#) [[↗](#)] — (IBM) Filed 3 May 1988, granted 12 June 1990 (Now expired)
- [U.S. Patent 4,935,882](#) [[↗](#)] — (IBM) Filed 20 July 1988, granted 19 June 1990 (Now expired)
- [U.S. Patent 4,989,000](#) [[↗](#)] — Filed 19 June 1989, granted 29 January 1991 (Now expired)
- [U.S. Patent 5,099,440](#) [[↗](#)] — (IBM) Filed 5 January 1990, granted 24 March 1992 (Now expired)
- [U.S. Patent 5,272,478](#) [[↗](#)] — (Ricoh) Filed 17 August 1992, granted 21 December 1993 (Now expired)

Note: This list is not exhaustive. See the following link for a list of more patents.^[4] The [Dirac codec](#) uses arithmetic coding and is not patent pending.^[5]

Patents on arithmetic coding may exist in other jurisdictions, see [software patents](#) for a discussion of the patentability of software around the world.

Benchmarks and other technical characteristics [[edit](#)]

Every programmatic implementation of arithmetic encoding has a different compression ratio and performance. While compression ratios vary only a little (usually under 1%),^[6] the code execution time can vary by a factor of 10. Choosing the right encoder from a list of publicly available encoders is not a simple task because performance and compression ratio depend also on the type of data, particularly on the size of the alphabet (number of different symbols). One of two particular encoders may have better performance for small alphabets while the other may show better performance for large alphabets. Most encoders have limitations on the size of the alphabet and many of them are specialised for alphabets of exactly two symbols (0 and 1).

See also [[edit](#)]

- [Data compression](#)

Notes [\[edit\]](#)

1. [^] [\[1\]](#) [What is JPEG? comp.compression Frequently Asked Questions \(part 1/3\)](#)
2. [^] ["Recommendation T.81 \(1992\) Corrigendum 1 \(01/04\)"](#) [↗](#). *Recommendation T.81 (1992)*. International Telecommunication Union. 9 November 2004. Retrieved 3 February 2011.
3. [^] [JPEG Still Image Data Compression Standard](#), W. B. Pennebaker and J. L. Mitchell, Kluwer Academic Press, 1992. ISBN 0-442-01272-1
4. [^] [\[2\]](#) [comp.compression Frequently Asked Questions \(part 1/3\)](#)
5. [^] [\[3\]](#) [Dirac video codec 1.0 released](#)
6. [^] For instance, [Howard & Vitter \(1994\)](#) discuss versions of arithmetic coding based on real-number ranges, integer approximations to those ranges, and an even more restricted type of approximation that they call binary quasi-arithmetic coding. They state that the difference between real and integer versions is negligible, prove that the compression loss for their quasi-arithmetic method can be made arbitrarily small, and bound the compression loss incurred by one of their approximations as less than 0.06%. See: Howard, Paul G.; Vitter, Jeffrey S. (1994), "Arithmetic coding for data compression" [↗](#) (PDF), *Proceedings of the IEEE* **82** (6): 857–865, doi:10.1109/5.286189 [↗](#).



References [\[edit\]](#)

- [MacKay, David J.C.](#) (September 2003). "Chapter 6: Stream Codes". *Information Theory, Inference, and Learning Algorithms* [↗](#) (PDF/PostScript/DjVu/LaTeX). Cambridge University Press. ISBN 0-521-64298-1. Archived [↗](#) from the original on 22 December 2007. Retrieved 2007-12-30.
- [Press, WH; Teukolsky, SA; Vetterling, WT; Flannery, BP](#) (2007). "Section 22.6. Arithmetic Coding" [↗](#). *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). New York: Cambridge University Press. ISBN 978-0-521-88068-8.
- [Rissanen, Jorma](#) (May 1976). "Generalized Kraft Inequality and Arithmetic Coding" [↗](#) (PDF). *IBM Journal of Research and Development* **20** (3): 198–203. doi:10.1147/rd.203.0198 [↗](#). Retrieved 2007-09-21.
- [Rissanen, J.J.; Langdon, G.G., Jr](#) (March 1979). "Arithmetic coding" [↗](#) (PDF). *IBM Journal of Research and Development* **23** (2): 149–162. doi:10.1147/rd.232.0149 [↗](#). Archived [↗](#) (PDF) from the original on 28 September 2007. Retrieved 2007-09-22.
- [Witten, Ian H.; Neal, Radford M.; Cleary, John G.](#) (June 1987). "Arithmetic Coding for Data Compression" [↗](#) (PDF). *Communications of the ACM* **30** (6): 520–540. doi:10.1145/214762.214771 [↗](#). Archived [↗](#) (PDF) from the original on 28 September 2007. Retrieved 2007-09-21.
- [Rodionov Anatoly, Volkov Sergey](#) (2010) "p-adic arithmetic coding" Contemporary Mathematics Volume 508, 2010 Contemporary Mathematics
- [Rodionov Anatoly, Volkov Sergey](#) (2007) " p-adic arithmetic coding", <http://arxiv.org/abs//0704.0834v1> [↗](#)

External links [\[edit\]](#)

- [Black, Paul E.](#) "arithmetic coding" [↗](#). *Dictionary of Algorithms and Data Structures*. NIST.
- [Newsgroup posting](#) [↗](#) with a short worked example of arithmetic encoding (integer-only).
- [PlanetMath article on arithmetic coding](#) [↗](#)
- [Anatomy of Range Encoder](#) [↗](#) The article explains both range and arithmetic coding. It has also code samples for 3 different arithmetic encoders along with performance comparison.
- [Introduction to Arithmetic Coding](#) [↗](#). 60 pages.
- [Eric Bodden, Malte Clasen and Joachim Kneis: Arithmetic Coding revealed](#) [↗](#). Technical Report 2007-5, Sable Research Group, McGill University.
- [Arithmetic Coding + Statistical Modeling = Data Compression](#) [↗](#) by Mark Nelson.
- [Data Compression With Arithmetic Coding](#) [↗](#) by Mark Nelson (2014)

v · t · e		Data compression methods	[hide]
Lossless	Entropy type	Unary · Arithmetic · Golomb · Huffman (Adaptive · Canonical · Modified) · Range · Shannon · Shannon–Fano · Shannon–Fano–Elias · Tunstall · Universal (Exp-Golomb · Fibonacci · Gamma · Levenshtein)	
	Dictionary type	Byte pair encoding · DEFLATE · Lempel–Ziv (LZ77 / LZ78 (LZ1 / LZ2) · LZJB · LZMA · LZO · LZRW · LZS · LZSS · LZW · LZWL · LZX · LZ4 · Statistical)	
	Other types	BWT · CTW · Delta · DMC · MTF · PAQ · PPM · RLE	
Audio	Concepts	Bit rate (average (ABR) · constant (CBR) · variable (VBR)) · Companding · Convolution · Dynamic range · Latency · Nyquist–Shannon theorem · Sampling · Sound quality · Speech coding · Sub-band coding	

	Codec parts	A-law · μ -law · ACELP · ADPCM · CELP · DPCM · Fourier transform · LPC (LAR · LSP) · MDCT · Psychoacoustic model · WLPC
Image	Concepts	Chroma subsampling · Coding tree unit · Color space · Compression artifact · Image resolution · Macroblock · Pixel · PSNR · Quantization · Standard test image
	Methods	Chain code · DCT · EZW · Fractal · KLT · LP · RLE · SPIHT · Wavelet
Video	Concepts	Bit rate (average (ABR) · constant (CBR) · variable (VBR)) · Display resolution · Frame · Frame rate · Frame types · Interlace · Video characteristics · Video quality
	Codec parts	Lapped transform · DCT · Deblocking filter · Motion compensation
Theory	Entropy · Kolmogorov complexity · Lossy · Quantization · Rate–distortion · Redundancy · Timeline of information theory	
 Compression formats ·  Compression software (codecs)		

Categories: [Lossless compression algorithms](#)

This page was last modified on 2 September 2015, at 02:48.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

