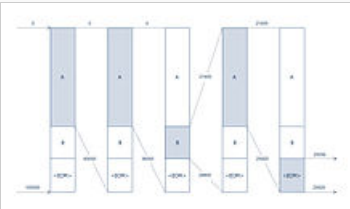
Range encoding

From Wikipedia, the free encyclopedia

Range encoding is an entropy coding method defined by G. Nigel N. Martin in a 1979 paper,[1] which effectively rediscovered the FIFO arithmetic code first introduced by Richard Clark Pasco in 1976.[2] Given a stream of symbols and their probabilities, a range coder produces a space efficient stream of bits to represent these symbols and, given the stream and the probabilities, a range decoder reverses the process.

Range coding is very similar to arithmetic encoding, except that encoding is done with digits in any base, instead of with bits, and so it is faster when using larger bases (e.g. a byte) at small cost in compression efficiency.[3] After the expiration of the first (1978) arithmetic coding patent,[4] range encoding appeared to clearly be free of patent encumbrances. This particularly drove interest in the technique in the open source community. Since that time, patents on various well-known arithmetic coding techniques have also expired.

How range encoding works [edit]



Graphical representation of the encoding process. The message being encoded here is "AABA<EOM>"

Range encoding conceptually encodes all the symbols of the message into one number, unlike Huffman coding which assigns each symbol a bit-pattern and concatenates all the bit-patterns together. Thus range encoding can achieve greater compression ratios than the one-bit-per-symbol lower bound on Huffman encoding and it does not suffer the inefficiencies that Huffman does when dealing with probabilities that are not exact powers of two.

The central concept behind range encoding is this: given a large-enough range of integers, and a probability estimation for the symbols, the initial range can easily be divided into sub-ranges whose sizes are proportional to the probability of the symbol they represent. Each symbol of the message can then be encoded in turn, by reducing the current range down to just that sub-range which corresponds to the next symbol to be encoded. The decoder must have the same probability estimation the encoder used, which can either be sent in advance, derived from already transferred data or be part of the compressor and decompressor.

When all symbols have been encoded, merely identifying the sub-range is enough to communicate the entire message (presuming of course that the decoder is somehow notified when it has extracted the entire message). A single integer is actually sufficient to identify the sub-range, and it may not even be necessary to transmit the entire integer; if there is a sequence of digits such that every integer beginning with that prefix falls within the sub-range, then the prefix alone is all that's needed to identify the sub-range and thus transmit the message.

Example [edit]

Suppose we want to encode the message "AABA<EOM>", where <EOM> is the end-of-message symbol. For this example it is assumed that the decoder knows that we intend to encode exactly five symbols in the base 10 number system (allowing for 10^5 different combinations of symbols with the range [0, 100000)) using the probability distribution {A: .60; B: .20; <EOM>: .20}. The encoder breaks down the range [0, 100000) into three subranges:

```
A:     [     0,  60000)
B:     [ 60000,  80000)
<EOM>: [ 80000, 100000)
```

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
Deutsch
Español
Français
한국어
日本語
中文
Edit links

Since our first symbol is an A, it reduces our initial range down to [0, 60000). The second symbol choice leaves us with three sub-ranges of this range, we show them following the already-encoded 'A':

```
AA:     [     0,  36000)
AB:     [ 36000,  48000)
A<EOM>: [ 48000,  60000)
```

With two symbols encoded, our range is now [0, 36000) and our third symbol leads to the following choices:

```
AAA:     [     0,  21600)
AAB:     [ 21600,  28800)
AA<EOM>: [ 28800,  36000)
```

This time it is the second of our three choices that represent the message we want to encode, and our range becomes [21600, 28800). It may look harder to determine our sub-ranges in this case, but it is actually not: we can merely subtract the lower bound from the upper bound to determine that there are 7200 numbers in our range; that the first 4320 of them represent 0.60 of the total, the next 1440 represent the next 0.20, and the remaining 1440 represent the remaining 0.20 of the total. Adding back the lower bound gives us our ranges:

```
AABA:     [21600, 25920)
AABB:     [25920, 27360)
AAB<EOM>: [27360, 28800)
```

Finally, with our range narrowed down to [21600, 25920), we have just one more symbol to encode. Using the same technique as before for dividing up the range between the lower and upper bound, we find the three sub-ranges are:

```
AABAA:     [21600, 24192)
AABAB:     [24192, 25056)
AABA<EOM>: [25056, 25920)
```

And since <EOM> is our final symbol, our final range is [25056, 25920). Because all five-digit integers starting with "251" fall within our final range, it is one of the three-digit prefixes we could transmit that would unambiguously convey our original message. (The fact that there are actually eight such prefixes in all implies we still have inefficiencies. They have been introduced by our use of base 10 rather than base 2.)

The central problem may appear to be selecting an initial range large enough that no matter how many symbols we have to encode, we will always have a current range large enough to divide into non-zero sub-ranges. In practice, however, this is not a problem, because instead of starting with a very large range and gradually narrowing it down, the encoder works with a smaller range of numbers at any given time. After some number of digits have been encoded, the leftmost digits will not change. In the example after encoding just three symbols, we already knew that our final result would start with "2". More digits are shifted in on the right as digits on the left are sent off. This is illustrated in the following code:

```
int low = 0;
int range = 100000;

void Run()
{
 Encode(0, 6, 10); // A
 Encode(0, 6, 10); // A
 Encode(6, 2, 10); // B
 Encode(0, 6, 10); // A
 Encode(8, 2, 10); // <EOM>
}

void Encode(int start, int size, int total)
{
 // adjust the range based on the symbol interval
 range /= total;
 low += start * range;
 range *= size;
```

```
 // check if left-most digit is same throughout range
 while (low / 10000 == (low + range) / 10000)
 {
  EmitDigit();
  range = (range % 10000) * 10;
 }
}

void EmitDigit()
{
 Console.Write(low / 10000);
 low = (low % 10000) * 10;
}
```

To finish off we may need to emit a few extra digits. The top digit of `low` is probably too small so we need to increment it, but we have to make sure we don't increment it past `low+range`. So first we need to make sure `range` is large enough.

```
// emit final digits
while (range < 10000)
{
 EmitDigit();
 range *= 10;
}

low += 10000;
EmitDigit();
```

One problem that can occur with the `Encode` function above is that `range` might become very small but `low` and `low+range` still have differing first digits. This could result in the interval having insufficient precision to distinguish between all of the symbols in the alphabet. When this happens we need to fudge a little, output the first couple of digits even though we might be off by one, and re-adjust the range to give us as much room as possible. The decoder will be following the same steps so it will know when it needs to do this to keep in sync.

```
// this goes just before the end of Encode() above
if (range < 1000)
{
 EmitDigit();
 EmitDigit();
 range = 100000 - low;
}
```

Base 10 was used in this example, but a real implementation would just use binary, with the full range of the native integer data type. Instead of `10000` and `1000` you would likely use hexadecimal constants such as `0x1000000` and `0x10000`. Instead of emitting a digit at a time you would emit a byte at a time and use a byte-shift operation instead of multiplying by 10.

Decoding uses exactly the same algorithm with the addition of keeping track of the current `code` value consisting of the digits read from the compressor. Instead of emitting the top digit of `low` you just throw it away, but you also shift out the top digit of `code` and shift in a new digit read from the compressor. Use `AppendDigit` below instead of `EmitDigit`.

```
int code = 0;

void InitializeDecoder()
{
        AppendDigit();
        AppendDigit();
        AppendDigit();
        AppendDigit();
        AppendDigit();
}
```

```
void AppendDigit()
{
        code = (code % 10000) * 10 + ReadNextDigit();
 low = (low % 10000) * 10;
}
```

In order to determine which probability intervals to apply, the decoder needs to look at the current value of `code` within the interval [low, low+range) and decide which symbol this represents.

```
int GetValue(int total)
{
        return (code - low) / (range / total);
}
```

For the AABA<EOM> example above, this would return a value in the range 0 to 9. Values 0 through 5 would represent A, 6 and 7 would represent B, and 8 and 9 would represent <EOM>.

Relationship with arithmetic coding [edit]

Arithmetic coding is the same as range encoding, but with the integers taken as being the numerators of fractions. These fractions have an implicit, common denominator, such that all the fractions fall in the range [0,1). Accordingly, the resulting arithmetic code is interpreted as beginning with an implicit "0.". As these are just different interpretations of the same coding methods, and as the resulting arithmetic and range codes are identical, each arithmetic coder is its corresponding range encoder, and vice versa. In other words, arithmetic coding and range encoding are just two, slightly different ways of understanding the same thing.

In practice, though, so-called range *encoders* tend to be implemented pretty much as described in Martin's paper,[1] while arithmetic coders more generally tend not to be called range encoders. An often noted feature of such range encoders is the tendency to perform renormalization a byte at a time, rather than one bit at a time (as is usually the case). In other words, range encoders tend to use bytes as encoding digits, rather than bits. While this does reduce the amount of compression that can be achieved by a very small amount, it is faster than when performing renormalization for each bit.

See also [edit]

- Multiscale Electrophysiology Format
- Shannon–Fano coding

References [edit]

1. ^ [a] [b] G. Nigel N. Martin, *Range encoding: An algorithm for removing redundancy from a digitized message*, Video & Data Recording Conference, Southampton, UK, July 24–27, 1979.
2. ^ "Source coding algorithms for fast data compression" Richard Clark Pasco, Stanford, CA 1976
3. ^ "On the Overhead of Range Coders", Timothy B. Terriberry, Technical Note 2008
4. ^ U.S. Patent 4,122,440 — (IBM) Filed March 4, 1977, Granted 24 October 1978 (Now expired)

External links [edit]

- Range Encoder
- "Range coder" by Arturo Campos
- "Anatomy of Range Encoder" by Andrew Polar

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