Introduction to Agnentropy

an entropy metric more accurate than Shannon entropy, based on provably invertible adaptive arithmetic encoding from minimal assumptions

Russell Leidich
https://agnentropy.blogspot.com

May 11, 2017

Keywords: agnentropy, agnostic, entropy, universal, code, encoding, arithmetic, compression, combinatorial, blockchain

0. Abstract

Claude Shannon[1] devised a way to quantify the information entropy[2] of a finite integer set, given the probabilities of finding each integer in the set. Information entropy, hereinafter simply "entropy", refers to the number of bits required to encode some such set in a given numerical base (usually binary). Unfortunately, his formula for the "Shannon entropy" seems to have been widely misappropriated as a means by which to measure the entropy of such sets by supplanting the probability coefficients (which are generally unknowable) with the normalized frequencies of the integers as they actually occur in the set. This practice is so common that Shannon entropy is often defined in precisely this manner, and indeed this is how we define it here. However, the inaccuracy induced by this compromise may lead to erroneous conclusions, especially when very short or faint signals are concerned. To make matters worse, the numerical behavior of Shannon entropy formula is rather unstable over large sets, where otherwise it would be more accurate.

Herein we introduce the concept of agnentropy, short for "agnostic entropy", in the sense of an entropy metric which begins with *almost* no assumptions about the set under analysis. (Technically, it's a "divergence" -- essentially a

Kullback-Leibler divergence[3] without the implicit singularies -- because it fails the triangle inequality. We refer to it as a "metric" only in the qualitative sense that it measures something.) This stands in stark contrast to the (compromised) Shannon entropy, which presupposes that the frequencies of integers within a given set are already known. In addition to being more accurate when used appropriately, agnentropy is also more numerically stable and faster to compute than Shannon entropy.

To be precise, Shannon entropy does not measure the number of bits in an invertibly compressed code. It is, more accurately, an underestimation of that value. Unfortunately, the margin of underestimation is not straightforwardly computable, and has a size O(Z), where Z is the number of unique integers in the set, assuming that said integers are of predetermined maximum size. By contrast, agnentropy underestimates that bit count by no more than 2, plus the size of 2 logplexes. (Logplexes are universal (affine) codes introduced in [8].) In practice, this overhead amounts to tens of bits, as opposed to potentially thousands of bits for Shannon. This difference has meaningful ramifications for the optimization of both lossless and lossy compression algos.

1. Distributional Encoding Algos

We use the term "distributional encoding algo" to refer to a lossless compression method, that is, a method by which a set of integers is transformed into bitstring, which can subsequently be transformed back into that set, identically. This transform is subject to the assumption that, while the integers may have arisen with different probabilities, no other aspects of the set are nonrandom in origin. In other words, the set is not expected to exhibit any contextual statistical associations apart from what would be expected to arise randomly as a result of the biased distribution.

In practice, we can equivalently think of such sets as being composed of unsigned integers on the (closed) interval [0, (Z-1)], not all of which *necessarily* present, but all of which *possibly* present, to the best of our prior knowledge. And for our purposes here, it's assumed that (Z>1). (If some values in the middle of the range are precluded, then a simple many-to-one remapping, to which we refer as "densification", can shrink Z.) We refer to values on this interval as "masks", where Z is the "mask span". And for the sake of consistency with source code, we refer to such integer sets as "mask lists" -- a "list" being the equivalent of a "set". The number of masks in the list is given by Q, the "mask count", which must be nonzero. The number of times which mask M occurs in the list is given by F(M), which is its "frequency". The number of times that a given frequency occurs in the list is H(F), its "population". (Population is just the frequency of the frequency, but we use different terms to avoid confusion.)

Computationally, it's easier to deal in units of nats (bits times (ln 2)) than bits. Hereinafter all entropy metrics will be expressed in nats. To begin with, we state the Shannon entropy S of a list of masks M with corresponding frequencies F(M), mask count Q, and mask span Z, in nats:

$$S \equiv Q \ln Z - \sum_{M=0}^{Z-1} F(M) \ln F(M)$$

where we refer to (Q ln Z) as the "raw entropy". The raw entropy is the theoretical minimum size of the set in nats, given no information other than Q and Z. Notice that the canonical form of this formula given in [2] has been modified to reflect *frequency* in place of *probability*; in other words, it's compromised in the manner discussed in the abstract. The reason is that probability is a function of the (unknowable) environment, so in practice we're forced to presume that

$$P(M) \equiv \frac{F(M)}{Q}$$

but unfortunately F(M) is unknown ahead of time. Therein lies the main reason why S is inaccurate: it disregards the potentially substantial overhead required to specify F(M) for all M, especially if F(M) is sparse (mostly zeroes).

More broadly, a table is provided below which summarizesis various distributional encoding algos along with their required overhead.

1.1. Table of Distributional Encoding Algos

Algo	**Required Overhead in Addition to the Mask List**
Raw entropy	Q. (Z is assumed to be 2.) Nats out equals nats in.
Huffman[4]	Q, Z, all F[M]. Maps each mask to a unique bitstring, resulting in a compressed size which is asymptotically a constant factor (greater than one) as big as S.
Arithmetic[5]	Q, Z, all F[M]. A more efficient algo than Huffman which uses nested fractions instead of a fixed mapping of masks to bitstrings.
Adaptive arithmetic[6]	Q, Z. F[M] is dynamically approximated, trading worse coding efficiency early on for the cost of storing F[M] explicitly. Simpler to implement but slower than conventional arithmetic encoding.
Combinatorial	Q, Z, H[F]. This algo encodes the masks using combinatorics, resulting in an integer of size proportional to logfreedom(Q, Z, H). Apparently, it has never actually been implemented, except for the case where (Z=2). For its part, logfreedom is precisely computable with Dyspoissometer[7].
Agnentropic	Only Q and Z, which as in other cases can be represented by 2 logplexes. F[M] is dynamically approximated. The resulting bitstring tends to be smaller than with most arithmetic encoding implementations. But while computing agnentropy costs O(Q) (essentially realtime), agnentropic encoding costs O(Q^2). This is actually a good thing for folks in need of a good proof-of-work function; it's not intended to serve as a practical compression algo. Nevertheless, Agnentro[9] can perform agnentropic encoding, in order to prove that it's invertible -- and so much more!
Superagnentropic	Absolutely nothing. Z expands dynamically over time, as new masks M are encoded as logplex(M+1) preceded by a "new" code (always zero), whose probability is also dynamically adjusted. The first "new" code is implicit

	and omitted from the stream. The stream ends with 2 "new" codes in a row. In theory, this method is a more accurate metric than agnentropy by a tiny margin in exachange for fantastic complexity due to the mask symmetry-breaking effect of "new". No codec or mathematical formalization is known to exist.

2. Agnentropic Encoding in Theory and Practice

There is little reason to make use of an entropy metric if it doesn't accurately approximate the number of bits or nats required to encode a mask list *invertibly*. As mentioned above, I created the Agnentro toolkit to, among other things, demonstrate that agnentropic encoding is invertible, given only Q and Z. Moreover, it's *unambiguous* in the sense that any random bits may follow the agnentropic code without affecting the decoded mask list. And finally, it's *dense* in the sense that there is no infinite series of bits which cannot be agnentropically decoded to an infinite mask list.

Here is how it works:

2.1. Givens: What Agnentropy *Doesn't* Account For

Apart from the mask list itself, Q and Z are the only givens to the agnentropic encoding algo. In practice, Z is probably known to the programmer way ahead of time (typically, a power of 2), and Q is specified by the operating system as a file size. But in a Turing machine, both could be specified by logplexes. Agnentro File, part of the Agnentro toolkit, can actually compute and store the logplexes prior to the agnentropic code corresponding to any arbitrary input file; it can then invert the encoding back to the original file.

2.2. Agnostic Frequency

Agnentropic codes rely on the concept of "agnostic frequency", as distinct from plain old frequency. While the frequency F(M) of every mask M in a null mask list is by definition zero, the *agnostic* frequency (F(M)+1) of every such mask is by definition one. In other words, we begin with 2 minimal assumptions:

1. The probability, and thus frequency, of all masks is identical.

2. The frequency of all masks less than Z is positive but as small as possible, hence one. The frequency of all masks greater than (Z-1) is zero (because they cannot occur).

We can actually assume even less, without resorting to the affine domain of superagnentropy, namely that the frequency of all masks is initially (1/Z); or equivalently that, while it's still one, *actual* frequency is to be multiplied by a factor of Z. Either way, however, the result is troublesome from a precision standpoint. To the extent that some cheap preprocessing can be used to reduce Z, this approach isn't worth the complexity. However, this does explain why, in certain short mask lists heavily dominated by a small minority of possible masks, Shannon entropy may be a more useful, if slower, metric.

2.3. Asymptotic Generator Discovery

All Z masks start with agnostic frequency one. The agnostic frequency of each mask M thus ends up as (F(M)+1) after the entire mask list has been processed, at which time the sum of all agnostic frequencies will be (Q+Z). Therefore:

$$\frac{F(M)+1}{Q+Z} \rightarrow P(M)$$

which is to say that the normalized agnostic frequency of mask M asymptotically converges to the probability with which the "generator" instantiates M. The generator is the unknowable physical system which gave rise to the mask list, which can nevertheless be modelled with asymptotic accuracy, assuming that its only statistical biases are distributional -- not contextual -- in nature. That is, the unknown generator G({0, 1... (Z-1)}) consists of Z *analog* probabilities P(M) which sum to one. G manifests itself with asymptotic accuracy in the limit that Q approaches infinity.

In practice, overlapping contiguous subparts of neighboring masks in order to create "virtual masks" can facilitate some understanding of the generator's *contextual* properties, if they exist to an extent beyond what distributional biases would imply, but this is a hack unrelated to agnentropy theory. (And again, by definition, G is assumed to be purely distributional.) Agnentro Find and Agnentro Scan offer optional mask overlap for this purpose.

2.4. Floors and Levels

2.4.1. The Floor Forumla

We define the "floor" D_{Z+P}(M) of mask M as the sum of the agnostic frequencies of all lesser masks as measured at zero-based index V of the mask list, and *before* reading the mask at that index:

$$D_Z(M) \equiv M$$

$$D_{Z+V}(M) \equiv M + \sum_{J=0}^{V-1} (U_J < M), V > 0$$

where U_J is the mask at index J of the mask list U, and the term "(U_J<M)" is one if U_J is less than M, else zero.

2.4.2. Levels

As a consequence of the floor formula, each mask M at index V of the mask list "owns" a contigous set of whole numbers, called "levels". The number of levels owned by M is just (F(M)+1), as evaluated *before* reading the mask at that index. This set is known as the "level allocation" of the mask. We refer to the least such level as the "floor" of M, and the greatest as its "ceiling".

2.4.3. Floor Lists

From the foregoing "floor formula", we can make a "floor list" which maps each mask M to its floor. (In practice, this is best implemented as a binary tree, but notionally, it's just a list.) Initially, because all agnostic frequencies are one, the floor of M is the only level owned by M, and simply has the value M. But suppose we have a mask list consisting of Q masks:

$$U \equiv \{U_0, U_1, \ldots U_{Q-1}\}$$

which will be processed from left to right. Now, after U_0 has been processed, the floor list looks like this, reflecting its new agnostic frequency of 2:

Mask=M	**Floor=D_{Z+1}(M)**
0	0
1	1
...	...
U_0	U_0
U_0+1	U_0+2
...	...
Z-1	Z

Therefore, the floor of M is still M for all (M<=U_0); for all other values of M, it's (M+1).

Now suppose (U_1 >U_0). Then the floor list morphs as follows:

Mask=M	**Floor=D_{Z+2}(M)**
0	0
1	1
...	...
U_0	U_0
U_0+1	U_0+2
...	...
U_1	U_1+1
U_1+1	U_1+3

...	...
Z-1	Z+1

...and finally suppose ($U_2 = U_0$):

Mask=M	**Floor=D_{Z+3}(M)**
0	0
1	1
...	...
U_0	U_0
U_0+1	U_0+3
...	...
U_1	U_1+2
U_1+1	U_1+4
...	...
Z-1	Z+2

Remember that D_{Z+3} is merely the state of the floor list as it existed after reading U_2. Therefore, while it will effect the probability distribution of U_3, it had no such effect on U_2.

2.5. Protoagnentropic Codes

Given Q and Z, and having set all F(M) to zero, we begin to load masks from U one at a time. These masks are combined in an iterative manner to produce a "protoagnentropic code", which is a whole number which can be inverted back to U. Protoagnentropic codes are so named because they're the progenitors to "agnentropic codes", which we'll investigate later.

2.5.1 The Envelope Pochhammer

As we'll confirm below, the number K of unique protoagnentropic codes is given by:

$$K \equiv N(Z,Q) \equiv Z*(Z+1)*(Z+2)*\ldots*(Z+Q-1)$$

which is simply the product over V from zero to (Q-1) of (the number of levels that exist before reading mask U_V). The quantity N is known as a "Pochhammer" in the WolframAlpha sense:

$$N(Z,Q) \equiv Pochhammer(Z,Q)$$

We refer to K as the "envelope Pochhammer" because it envelops all possible protoagnentropic codes. Again, Q is required to be nonzero. However, we explicitly define N(Z, 0) to be one, consistent with the definition of a Pochhammer.

2.5.2 The Protoagnentropic Encoding Algo

Encoding itself begins with a single mask, U_0, which results in a protoagnentropic code, B({U_0}), of identically the same value:

$$B(\{U_0\}) \equiv U_0$$

We can then append U_1 as follows:

$$B(\{U_0, U_1\}) \equiv U_0*(Z+1)+D_{Z+1}(U_1)$$

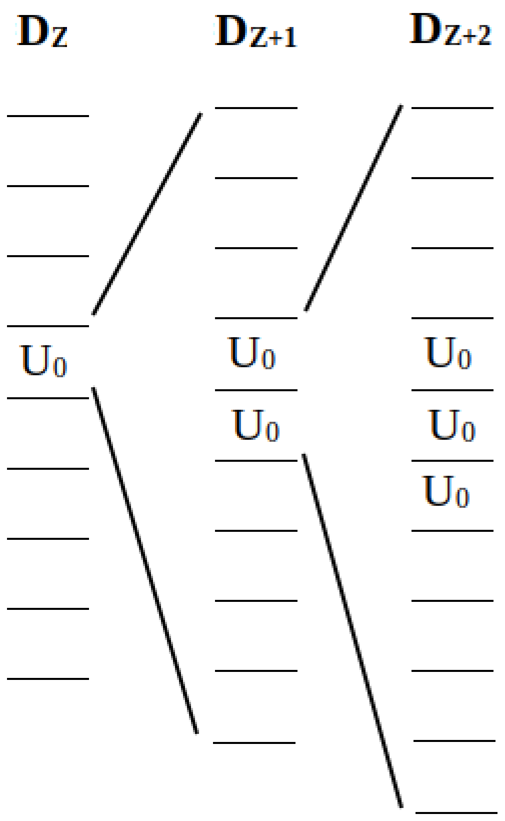
which is clearly invertible by dividing by (Z+1), which would yeild quotient U_0 and remainder $D_{Z+1}(U_1)$. The identity holds even if ($U_1 = U_0$). Just remember that we're dealing in floors, not masks, starting with the second addend. But what if we have {U_0, U_1, U_2}, where all the masks are unique? In that case:

$$B(\{U_0, U_1, U_2\}) \equiv U_0*(Z+1)*(Z+2)+D_{Z+1}(U_1)*(Z+2)+D_{Z+2}(U_2)$$

As before, at each step, we multiply the existing expression by the number of levels that exist before reading the next mask. And notice that the 2 (D)s are not the same: one refers to a floor list involving (Z+1) levels, whereas the other involves (Z+2). We can iterate this process indefinitely and still produce an invertible code. But what if some of the codes are equal? For example:

$$B(\{U_0, U_0, U_0\}) \equiv U_0 * (Z+1) * (Z+2) + D_{Z+1}(U_0) * (Z+2) + D_{Z+2}(U_0) * 2$$

(The floor functions have been shown for clarity, but in fact they all resolve to U_0 because no lesser masks are involved.) Now, what's that factor of 2 doing there? It's effectively cramming the high bit of the floor of the last U_0 back into the level allocation of the second U_0. We can do this without loss of invertibility because: (1) The first U_0 owns one of Z levels. (2) The 2nd U_0 has agnostic frequency 2, but cannot take advantage of this because all (Z+1) levels it inherited need to scale down to fit into the single floor owned by the first U_0. (3) The 3rd U_0 has agnostic frequency 3. We can double its floor because the (Z+2) levels it inherited from the 2nd U_0 need to scale down to fit into the 2 levels owned by the 2nd U_0. Thus the 2nd U_0 owns 2 levels, each consisting of (Z+2) sublevels. Before reading the 3rd U_0, all masks other than U_0 own a consecutive pair of those sublevels, while U_0 itself owns a triplet. Perhaps this is easier to understand graphically, where the levels owned by U_0 are marked as such; remember, the number of levels equals the agnostic frequency of U_0:



The bottom line is that the frequency multiplier lags the frequency by one mask. Let's consider another example, where ($U_0<U_1<U_2$):

$$\begin{gathered}B(\{U_0,U_0,U_0,U_1,U_0,U_2\})\equiv U_0*(Z+1)*(Z+2)*(Z+3)*(Z+4)*(Z+5)\\ +U_0*(Z+2)*(Z+3)*(Z+4)*(Z+5)\\ +U_0*2*(Z+3)*(Z+4)*(Z+5)\\ +(U_1+3)*2*3*(Z+4)*(Z+5)\\ +U_0*2*3*(Z+5)\\ +(U_2+5)*2*3*4\end{gathered}$$

Where (U_1+3) reflects the 3 lesser masks which precede it, and likewise for (U_2+5). We can generalize this as follows:

$$B(U)\equiv\sum_{J=0}^{Q-1}D_{Z+J}(U_J)*N(Z+J+1,Q-J-1)*\prod_{M=0}^{Z-1}(F'(M,J)!)$$

where $D_{Z+J}(U_J)$ and F'(M, J) are the floor of U_J and the (nonagnostic) frequency of M respectively, both of which computed before accounting for U_J itself. (And N is just a factor of the envelope Pochhammer.) Bear in mind that zero factorial is one, which is thus the lower bound of the product term. Given this, and the fact that $D_Z(U_0)$ can take one of Z states, the number of unique protoagnentropic codes is indeed N(Z, Q), which is just K, as stated above.

However, note that the number of unique mask lists is merely Z^Q, which at most equals K. This means that multiple protoagnentropic codes may invert to the same mask list. The formula for B given above produces a "canonical" protoagnentropic code, in the sense that it's the *least* such code which will invert to mask list U.

2.6. Protoagnentropic Span

The number of protoagnentropic codes -- canonical or not -- which invert to U is given by its "span", T:

$$T(U)\equiv(F(U_{Q-1})+1)*\prod_{M=0}^{Z-1}(F'(M,Q-1)!)\equiv\prod_{M=0}^{Z-1}(F(M)!)$$

which is just the previous product term evaluated when (J=(Q-1)), times the frequency of the last mask (including the last mask itself). The simplified version on the right assumes that F(M) includes all masks and is nonagnostic:

$$F(M)\equiv\sum_{J=0}^{Q-1}(U_J=M)$$

where (U_J=M) is one if U_J equals M, else zero.

Now, suppose we have the whole number U' expressed in base Z, the digits of which are: U_0 in the most significant position, U_1 in the 2nd most significant, etc., in exactly the order specified in the mask list U. Furthermore define B'(U') as a function returning the same value as B(U), but taking the equivalent whole U' as its input. Then, given that B is by definition canonical, it follows that:

$$T(U)\equiv B'(U'+1)-B'(U')$$

which, as stated, is an identity which holds so long as U' is the base-Z equivalent of U.

This implies that T(U) successive protoagnentropic codes, starting with the *canonical* protoagnentropic code B(U), all invert to U. Furthermore, because all level allocations are contiguous, there must exist some integer Y such that:

$$B'(U')\leq Y<B'(U'+1)$$

and

$$Y \bmod T(U)=0$$

In other words, we could transform B'(U') into a much smaller integer -- potentially even smaller than Z^Q -- by dividing Y by T(U), thereby accomplishing invertible distributional data compression on U. The trick is to do this in such a way that guarantees invertibility in some convenient base (usually binary).

Another consideration is that we want all, or all but a constant, of the bits of Y to remain the same if we append a new mask, U_J, to U. This requirement, which is important for the sake of dynamic expandability (think: blockchain), is fundamentally inconsistent with the manner in which Y is computed.

Alternatively, we could just save the numerator and denominator of the reduced fraction (B(U)/K). Along with a couple logplexes providing the bit counts, that fraction would preserve all of the information in B(U). We refer to its as the "agnentropic rational", R, of U:

$$R(U) \equiv \frac{B(U)}{K}$$

where K is the envelope Pochhammer defined previously. However, due to the vagaries of fraction reduction, this practice would lead to asymmetric compression performance in the sense that some mask lists would compress much better than others despite having equal population lists. So instead of saving R(U) literally, we approximate it as A(U), a binary fraction on [0, 1). Similarly to Y, it has the property that:

$$R(U) \leq A(U) < \frac{B'(U'+1)}{K}$$

A(U) is the "canonical agnentropic code" of U, which is the *shortest* (and, if not unique, *least*) binary fraction which inverts to U *regardless of the "junk" bits which follow it* (the "unambiguous invertibility" constraint). *All* binary fractions which unambigously invert to U, or U followed by other masks, are "agnentropic codes" of U.

But before we explain how to convert U to its A(U), we must precisely define our entropy metric, namely, agnentropy. We must then show that the size, in binary, of a canonical agnentropic code is bounded by the agnentropy of U plus some predictable (and hopefully small) overhead.

2.7. The Agnentropy Formula

We can place a lower bound on the amount of information required to be present in A(U) in order to allow it to invert umambiguously to U. We call this quantity the "agnentropy" of U, denoted X(U). Conceptually, it's proportional to the number of bits required to distinguish B(U) from the protoagnentropic codes which don't invert to U. It has units of nats, mainly because we don't want the extra precision loss due to carrying around factors of (ln 2):

$$X(U) \equiv \ln \frac{N(Z,Q)}{T(U)}$$

which, after substituting for N and T, is:

$$X(U) \equiv \ln \frac{Z*(Z+1)*(Z+2)*...*(Z+Q-1)}{\prod_{M=0}^{Z-1} (F(M)!)}$$

But the log of a product is just a sum of logs. Therefore:

$$X(U) \equiv \ln((Q+Z-1)!) - \ln((Z-1)!) - \sum_{M=0}^{Z-1} \ln(F(M)!)$$

First of all, note that X(U) is commutative with respect to mask order. In other words, agnentropy depends only on frequency. (Moreover, it depends only the *populations* of the frequencies, but the form above is often more computationally expedient.)

Now, on the face of it, X(U) is computationally expensive for a mask list of any reasonable size. That's true in the sense that we must first tally up all the masks of each type, resulting in F(M). However, that task takes essentially no longer than just reading the mask list from storage in the first place. But what about all those expensive log calculations?

In practice, because various masks tend to have the same frequency, a cache of previously used log results can greatly accelerate the math. (Agnentro Find and Agnentro Scan actually use this technique.) But it's better than that because we need not compute the log of a factorial as a sum of logs. There's a much faster method which takes advantage of the following identity:

$$\sum_{a=1}^{A} \ln a \equiv \ln(A!) \equiv log\Gamma(A+1)$$

In other words, we can evaluate the loggamma function once instead of the log function N times in order to compute log(N!):

$$X(U) \equiv log\Gamma(Q+Z) - log\Gamma(Z) - \sum_{M=0}^{Z-1} log\Gamma(F(M)+1)$$

which is the "agnentropy formula". Again, this formula could be expressed in terms of populations of frequencies, although I chose not to do that because tracking poplations is rather more computationally expensive. (The Poissocache library included with Agnentro does exactly that, but for logfreedom as opposed to agnentropy. It's actually a generic hash cache manager with miss-count-per-lookup expected to follow a Poisson distribution.)

By the way, one might reasonably ask whether it would be more useful to include the cost of encoding Q and Z, via logplexes or some other universal code, in the definition of agnentropy. The *practical* answer is no, simply because these values are usually constant: in real life software, we tend to deal with blocks of constant numbers of items, each of which some constant number of bits in size. Including the logplex overhead would at best waste time and tax precision.

It's uncanny how, having thus succintly expressed agnentropy in terms of the loggamma function, the frequencies have suddenly become agnostic. This is one of many weak hints which suggest to me that there's a connection between agnentropy and the Riemann zeta function. Another is that the sum of loggammas expands into a linear combination of logs of natural numbers, which in turn expands into a linear combination of logs of primes. But I'll leave it at that for now.

The loggamma, for its part, is a beast. Just the infinite series used to compute it requires a number of weird transcendentals and precomputed rational coefficients. To make matters worse, it's very difficult to see how terminating the series at any given term affects the error in the result. This is all to say that it's not readily apparent how to implement it using interval arithmetic.

But necessity is the mother of invention! First of all, I published in [10] a rigorous demonstration that the error is bounded by a simple expression. Then I implemented fast interval arithmetic libraries (in open source C, like all of Agnentro) called "Fracterval U64" and "Fracterval U128" (which are 64 and 128-bit normalized unsigned fixed-point interval libraries, respectively). Best of all, Agnentro doesn't use any floating-point math, which means it will execute identically on all supported platforms. (Floating-point functions suffer from a seemingly hopeless dearth of standardization, notwithstanding IEEE attempts to the contrary. This raises the specter of platform dependency and even security vulnerabilities. Besides, fractervals (normalized intervals) are simpler to understand and faster in many cases.) The result is that agnentropy can now be computed in a reproducible manner with interval math, such that the result is guaranteed to be between a certain minimum and maximum value. In practice, the uncertainty tends to be negligible compared to the input file size, which enables highly sensitive signal detection and comparison. More on that later.

As to performance, on a commodity 3 GHz Intel CPU in a single thread, the Agnentro File utility is able to compute the agnentropy of a billion bytes ($Q=10^9$, $Z=256$) in about a second, after subtracting read time from storage.

2.8. Computational Complexity

Asymptotically, it's quite clear from using Agnentro File that the computational complexity of agnentropy is O(Q), which means that the time it takes is asymptotically proportional to the size of the mask list. Granted, some time is required up front to lazily populate the log and loggamma lookup tables. Fortunately, those tables can be reused for multiple files in the case of bulk processing.

The story is very different for agnentropic *encoding*.

First of all, agnentropic codes are in no way commutative with respect to mask order. They can't be, because they're required to be invertible, which includes the preservation of order as well as frequency. In fact, they're quite expensive to compute, to the tune of $O(Q^2)$. Of this, one factor of O(Q) is due to handling each mask one by one, and another factor of O(Q) is due to the increasingly large integer multiplications and divisions required to compute the protoagnentropic code. (I wrote the open source Biguint library, provided free with Agnentro, just for the purpose of handling all this painful arithmetic.)

2.9. Beware the Arithmetic Gap!

For the record, arithmetic compression, whether or not adaptive, suffers from the same $O(Q^2)$ malaise, but in practice its complexity is reduced to something like O(Q ln Q) by implementing a close approximation of the algo which obviates the need for large integer arithmetic. However, this approximation method may create cases in which certain arithmetic codes do not decompress properly due to the programmer's failure to consider the gaps between valid cases, and in particular fractions very close to one. This "arithmetic gap problem" may well manifest as future security vulnerabilities. Personally, I would bet on it.

This brings us to the *advantage* of evaluating agnentropic codes as a practical matter: their execution *inefficiency* is rooted in their coding *efficiency*, so in principle they could give rise to a robust class of proof-of-work functions, of the sort used in cryptocurrency and other blockchain apps. At the same time, due to their dense nature, they would safely avoid the arithmetic gap problem.

2.10. Agnentropic Codes

As stated above, we define the canonical agnentropic code A(U) as the shortest (and, if not unique, least) binary fraction which will unambiguously invert to U. First of all, this constraint falls short of the mark for a universal code, like a logplex, because we still need Q and Z to be provided up front.

This is why agnentropy is entropy from *almost* nothing, as opposed to entropy from nothing.

We already know the minimum number of bits required to represent A(U): it's just the agnentropy over (ln 2), rounded up to the nearest bit. If this were not the case, then we'd have asymmetrical compression performance even in cases where R(U) were exactly representable in a finite bitstring (sequence of bits). And indeed, we can only reach the exact minimum in such cases.

But now let's consider the *maximum* number of bits which might be required to approximate R(U) as a binary fraction. Our precision must be sufficient to recover U from its contiguously owned subset of [0, 1). That interval has size (1/C(U)) where:

$$C(U) \equiv \frac{N(Z,Q)}{T(U)}$$

Now, C(U) is a natural number because N(Z,Q) is a nonzero multiple of the span. (This is true even though the same, in general, does not hold for B(U).) The reason is that the factors of N(Z, Q) increment at least as quickly as the frequency of the most frequent mask.

If C(U) is a power of 2, then ($\log_2$ C(U)) is the exact number of bits required to unambiguously invert B(U). Otherwise, things get complicated.

Consider the case where (Q=1) and (Z=3). B(U) is then either zero, (1/3), or (2/3). Because ($\log_2$ 3) is between one and 2, we should be able to invertibly represent each of these fractions in 2 bits. And indeed we can:

0/3 = 0.000000...
1/3 = 0.010101...
2/3 = 0.101010...

As you can see, we can ignore the zero to the left of the binary point because it's implied. The first 2 bits to the right then distinguish all 3 fractions from one another. But this is not sufficient to guarantee *unambiguous* invertibility.

Consider, for example, the case of 0.0100, which ends in a pair of junk zeroes. This represents the interval [(1/4), (5/16)), which falls in the bottom third of [0, 1), which means that it's "owned" by zero. So in practice we would need to encode 3 bits to the right of the binary point (0.011), which is [(3/8), (1/2)), which is fully owned by (1/3). However, in the case of (0/3), we could in fact use just 2 such bits (0.00), as it could never reach (1/3), regardless of the junk bits. This simple example illustrates why different values of U with the same agnentropy may nevertheless have canonical agnentropic codes which differ in size by up to one bit.

Granted, in the simple example above, we would know that 0.0100 is actually owned by (1/3) because (Q=1) implies that all frequencies must be either zero or one. But in practice, there's no easy way to know ahead of time where the fraction ends, even if we have Q and Z up front, because the frequencies which imply the span and thus the size of the fraction are discovered incrementally. It's not *impossible*, but learning that information would add fantastic complexity in the form of a binary search for the bit position at which the inverse of the inverse of the code equals the code itself -- all in order to save one bit. It's not worth the trouble, even for merely theoretical purposes.

This, in short, is why agnentropy times (ln 2) is at most 2 bits less than the number of bits in the agnentropic code: we need up to one bit to round up to the nearest whole bit, plus up to one additional bit to ensure unambiguous invertibility. Every test of Agnentro File has thus far produced results consistent with this constraint.

2.10.1. The Agnentropic Encoding Algo

Here, then, is how to produce an agnentropic code, given the whole number C(U) computed above. This code, first of all, must be fully owned by the following interval:

$$\left[\ R(U), \frac{B(U)+T(U)}{K}\ \right)$$

because the lower (closed) limit is the least rational which is invertible to U, and the upper (open) limit is the maximum such rational. We want to find the shortest (and, if not unique, least) possible binary fraction which fits this constraint and also has unambiguous invertibility.

1. Compute:

$$J \equiv \lfloor \log_2(C(U)) \rfloor + 2$$

that is, the integer floor of ($\log_2 C(U)$), plus 2. This is the maximum possible number of bits required. Conceptually, one bit is required for the rounding error in agnentropy conversion to binary, and another bit is sometimes required to compensate for odd level allocation alignment.

2. Compute:

$$A_L \equiv \frac{B(U) \ll J}{K} + !!((B(U) \ll J) \, mod \, K)$$

rounded down to the nearest whole number. (The "<<" symbol denotes shifting to the left, in this case by J bits. The "!!" symbol denotes "Booleanization", which returns zero for a zero operand, or else one.) The resulting value, which represents a fraction over 2^J, will be the *least* such fraction which is fully owned by the interval above, hence the subscript "L".

3. Compute:

$$A_G \equiv \frac{(B(U) + T(U)) \ll J}{K} - 1$$

rounded down to the nearest whole number. The resulting value, which represents a fraction over 2^J, will be the *greatest* such fraction which is fully owned by the interval above.

4. Set A_{L0}, A_{L1}, A_{G0}, and A_{G1} to bit zero of A_L, bit one of A_L, bit zero of A_G, and bit one of A_G, respectively. Use the following table to determine how to set A(U). For example A_G[1X] means "Set A(U) to A_G. Set bit one to one. Delete bit zero by shifting to the right by one." Then also subtract the number

of (X)s from J.

A_{L1}	A_{L0}	A_{G1}	A_{G0}	**A(U)**
0	0	0	0	A_L[00]
0	0	0	1	A_L[0X]
0	0	1	0	A_L[0X]
0	0	1	1	A_L[XX]
0	1	0	0	A_L[1X]
0	1	0	1	A_L[01]
0	1	1	0	A_L[01]
0	1	1	1	A_L[1X]
1	0	0	0	A_L[1X]
1	0	0	1	A_L[1X]
1	0	1	0	A_L[10]
1	0	1	1	A_L[1X]
1	1	0	0	A_L[11]
1	1	0	1	A_G[0X]
1	1	1	0	A_G[0X]
1	1	1	1	A_L[11]

5. Now that J is its minimum possible value, arrange all J bits of A(U), so that bit zero of memory contains bit (J-1) of A(U), bit one contains bit (J-2), etc. (Depending on whether one performed the foregoing computations in bitwise big or little endian, A(U) may already be in this form. Also, note that hexadecimal bytes are conventionally displayed with the high nybble first, whereas successive bytes involve progressively higher bits of A(U). This can be a source of confusion when inspecting memory contents, especially when, as in the case of Agnentro File output, A(U) is preceded by logplex codes for Q and Z.)

6. Save all J bits of A(U) without regard to the junk bits that follow them, but

beware that saving such bits straight out of memory may have security ramifications.

If all of this sounds complicated, feel free to see how Agnentro File does it. It also performs decoding, which we'll explore later.

2.10.2. Agnentropic Encoding Example

Find the agnentropic code of (U={3, 5, 3, 2, 1, 3, 3}) when (Q=7) and (Z=6).

First, compute the envelope Pochhammer:

$$K=(6+0)*(6+1)*(6+2)*(6+3)*(6+4)*(6+5)*(6+6)=3{,}991{,}680$$

Next, compute the protoagnentropic code:

$$\begin{gathered}B= \\ +3*(6+1)*(6+2)*(6+3)*(6+4)*(6+5)*(6+6) \\ +(5+1)*(6+2)*(6+3)*(6+4)*(6+5)*(6+6) \\ +3*(6+3)*(6+4)*(6+5)*(6+6) \\ +2*2*(6+4)*(6+5)*(6+6) \\ +1*2*(6+5)*(6+6) \\ +(3+2)*2*(6+6) \\ +(3+2)*2*3 \\ =2{,}607{,}414\end{gathered}$$

Compute the span:

$$T=(0!)*(1!)*(1!)*(4!)*(0!)*(1!)=24$$

Compute C:

$$C=K/T=3{,}991{,}680/24=166{,}320$$

Compute J:

$$J=\lfloor\log_2(166{,}320)\rfloor+2=19$$

Compute A_L in binary:

$$A_L = \frac{(2{,}607{,}414 \ll 19)}{3{,}991{,}680} + !!((2{,}607{,}414 \ll 19) mod\, 3{,}991{,}680) = 342{,}472 = 1010011100111001000$$

Compute A_G in binary:

$$A_G = \frac{((2{,}607{,}414 + 24) \ll 19)}{3{,}991{,}680} - 1 = 342{,}473 = 1010011100111001001$$

Examine the low bits:

$$A_{L1} = 0, A_{L0} = 0, A_{G1} = 0, A_{G0} = 1$$

Look them up in the table above, then issue the canonical agnentropic code accordingly, which turns out to contain 18 bits:

$$A = A_L[0X] = 101001110011100100$$

This answer has been verified with Agnentro File by creating a file equal to U and enabling automask (whereby the utility automatically computes Z). It says: "agnentropy_bit_count=12", which is hexadecimal for 18. It also mentions "output_bit_count=1C", which means that the low 10 bits of the output file are consumed by logplexes storing Q and Z. For that matter, another instance of Agnentro File was able to decode its own output file back to exactly U. Then, I appended more than 64 ones to the end of the canonical agnentropic code; Agnentro File still produced the same output, confirming that my junk bits didn't matter. Finally, I incremented the original output A, whereupon decoding failed to produce U, which confirmed that A does not contain any redundant information. (In some cases, incrementing A will still preserve U, due to violations of canonical constraints which may or may not ensure unambiguous invertibility. To be sure, such violations should not be allowed to create security vulnerabilities.)

2.11. Verifiability and Security Considerations

The beauty of agnentropic codes is that they can be inverted given only Q and Z; we need not know where they end. This is especially useful where dense packing is required, in which case they can be blindly concatenated one after another. This practice is safe because, after the first fraction is decoded, it can be reencoded in order to (1) determine the number of bits in the fraction just decoded, and therefore no longer needed; and (2) verify that the reencoded fraction matches the one that was read from the file, which, if it fails, suggests that an attacker may be attempting to exploit a security vulnerability by employing noncanonical agnentropic codes.

It's possible to expand the codes bit by bit, thereby appending more masks after the original code has been written to memory. However, this requires accumulating the frequency list over multiple sessions. It also may require modification of some bits of the fraction, due to the discretization of the terminal bits as discussed above. Nevertheless this feature may have some practical use with regards to proof-of-work functions, above and beyond what $O(Q^2)$ already provides.

Finally, Q need not be known exactly; a lower bound will suffice. In other words, because agnentropic codes are expressed as fractions, we can always correctly decode fewer masks than the fraction actually encodes. Of course, in this case, the verify-by-reencoding strategy won't work. On the plus side, this technique can be used to recover some portion of data from a corrupted stream.

2.12. Agnentropic Decoding

Given Q and Z, which in the case of Agnentro File are decoded from logplexes, we must first compute J, the maximum possible number of bits required in order to completely decode U. If in fact we lack J bits, we can simply append zeroes to the end. This will work properly *provided* that the encoder faithfully saved a complete agnentropic code of U, whether or not canonical. So, compute:

$$J \equiv \lfloor \log 2(K) \rfloor + 2$$

which is just the greatest possible number of bits which could be required to encode a U, because it assumes a span of one.

Now, bear in mind that the code is currently in bitwise big endian form, where the most significant bit resides at bit zero of memory and the least significant at bit (J-1). We'll heretofore just consider the code and the states into which it evolves as a whole number called the "bitstring". Notionally, its binary point is located just before bit zero. We will operate on the bitstring in a big endian manner, just like a student would handle an arithmetic computation on paper.

Create a floor list which identically maps level M to mask M, for all Z masks. This floor list can thus also be used to compute agnostic frequency.

Now, iterate as follows:

1. Multiply the bitstring by the current number of levels, initially Z, then (Z+1), etc. This will usually result in a code exceeding one. Notionally, its integer part will appear at negative bit positions not exceeding bit negative one.

2. Set L to this integer part of the code.

3. Use the floor list to determine which mask M owns this level L. (Agnentro File does this via an iterative binary search of a binary tree.)

4. Store M to the next mask index in the output U, initially at index zero.

5. Set F to the agnostic frequency of M.

6. Adjust the floor list in order to reflect incrementation of the agnostic frequency (F(M)+1), but don't actually increment F(M) just yet.

7. Set D to the floor of M.

8. Subtract D from the integer part of the bitstring.

9. Divide the bitstring by F(M), as though the former were a whole number with the binary point located just after bit (J-1). If the remainder is nonzero, then increment the bitstring, assuming the same binary point.

10. Revert to the assumption that the binary point is just before bit zero. If Q masks have not yet been produced, goto 1.

11. If it matters for security reasons, reencode U in order to test whether or not the input agnentropic code was in fact canonical.

3. Remarks

This paper exists solely to explain the math behind agnentropic encoding, which in turn provides a logical justification for the use of agnentropy as an entropy metric. Whether or not agnentropic encoding finds the suggested uses in proof-of-work functions, blockchain encoding, or other areas in which it matters to have only one way of doing something, useful applications of agnentropy have already been demonstrated with the Agnentro toolkit.

For its part, agnentropy is fast enough to process signals in real time, as they arrive from the environment. This allows the detection of bursts of anomalously low, high, or out-of-band entropy. Such anomalies could in turn trigger more meticulous automated analysis because they might result from unexpected or interesting phenomena. Likewise, signals can be compared and classified by simply dropping them into buckets which coincide with their relative agnentropy levels.

Due to its incremental nature which obviates the need to account for the size of a frequency list, agnentropy is usually more accurate than Shannon entropy, to the extent that all Z masks are actually possible. It's also faster to compute in dynamic regimes, in which one mask is being supplanted for another, over and over again. These advantages make it a useful tool for realtime signal analysis.

For more details, or to download the Agnentro toolkit, see https://agnentropy.blogspot.com .

Bibliography

[1] https://en.wikipedia.org/wiki/Claude_Shannon

[2] https://en.wikipedia.org/wiki/Entropy_%28information_theory%29

[3] https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

[4] https://en.wikipedia.org/wiki/Huffman_coding

[5] https://en.wikipedia.org/wiki/Arithmetic_coding

[6] https://en.wikipedia.org/wiki/Arithmetic_coding#Adaptive_arithmetic_coding

[7] https://dyspoissonism.blogspot.com

[8] https://vixra.org "Introduction to Logplex Encoding"

[9] http://agnentropy.blogspot.com

[10] http://vixra.org/abs/1609.0210

[11] http://cs.annauniv.edu/insight/Reading%20Materials/maths/algebra/indet/modkut.htm