HIGH DENSITY DNA MANIFOLDS WITH ARITHMETIC CODING

(Draft)

Joe Davis
25 April 2025

DNA Manifold approach to DNA information-keeping is tied to the idea that every DNA triplet or, 3-base codon holds 3 distinct "pages" of information. One page is the initial sequence of DNA bases. A second "page" consists of a selected numerical value assigned to respective codons translating for a particular amino acid ("Silent Code"), and a third "page" is made up of numbers assigned to each of 20 amino acids ("Amino Code).

Below is a practical, high-density encoding workflow – without any physics-defying trickery – that takes an English language phrase, compresses it near the Shannon limit (using Arithmetic coding), then encodes the compressed bitstream into a "Manifold" DNA design capable of ~3.5-4.0 bits/base. This preserves this notion of "pages" or "manifolds" (i.e., recursion bits, optional "silent" bits) but aims for maximal realistic information density well above ~3 bits/base.

ARITHMETIC CODING

Text input:

"THE LORD WHOSE ORACLE IS AT DELPHI NEITHER SPEAKS NOR CONCEALS BUT GIVES A SIGN" (Heraclitus Fragment)

- Gather character frequencies (upper case letters)

Building Arithmetic Encoder:

- Arithmetic coding compresses a sequence of characters (like "abc") into a single number between 0 and 1. This number uniquely represents the whole message based on the probabilities (CDFs) of each character.
- Unlike Huffman encoding with discrete bit patterns, arithmetic coding continuously "zooms in" on the probability interval of each symbol, achieving closer to the theoretical Shannon limit
- Typical savings resulting from Arithmetic vs. Huffman coding is ~5-10% bit reduction for small alphabets

Character Frequency Table:

- Calculate the probabilities and Cumulative Distribution Functions (CDF*) for each character based on their frequency in the Heraclitus fragment.

* The Cumulative Distribution Function (CDF) gives the probability that the random variable X is less than or equal to x and is usually denoted F(x)

Characters and their counts:

- Example: E = 6, and say, total characters = 58, so

  P(E)=6/58≈0.1034

This is a way of calculating the probability of event E happening.

- There are 6 outcomes that make event E happen, and 58 total possible outcomes.
- So, the probability of E is:

  favorable outcomes / total outcomes = 6/58 = approximately 0.1034

- The chance of event E happening is about 10.34%.

CDF Calculation:

This step compresses a sequence of characters (like "abc") into a single number between 0 and 1. This number uniquely represents the whole message based on the probabilities (CDFs) of each character.

- Calculate CDF by sorting characters by their probability and computing a running sum

Arithmetic Encoding Using These CDFs:

- Start with a range or interval from 0 to 1 — like a ruler that goes from 0.0 to 1.0 – a number line that starts at 0 and goes up to (but doesn't include) 1. This is the initial range.
- Go through the message one character at a time. For each character in the message:

  Shrink the current range to focus only on the part that corresponds to the character being encoded.

  This is done using CDF (Cumulative Distribution Function) values, which tell us the boundaries of each character's portion of the range.

For each character in the message the interval is narrowed down based on the probability of the character. This is where the CDF (Cumulative Distribution Function) comes in. Think of the CDF as a way of dividing up that 0 to 1 range into chunks for each possible character.

Update The Range

For each character:

- First, calculate the size of the current range (i.e., high - low).

- Then, use the CDF to find where this character falls in the range:

  high becomes: low + range * CDF (character)

  low becomes: low + range * CDF (previous character)
  (i.e., where the character's portion starts)

For example, if character B has a CDF of 0.5, and the previous character (A) has a CDF of 0.2, then B's portion of the range is between 20% and 50% of the current range.

After Processing All Characters:

The range [low, high) now contains the final encoded number. Any number in that range could represent the original message. Here the low value is used.

Convert Low to Binary:

Multiply low value by 2 repeatedly to extract each bit (just like converting a fraction to binary manually). This gives a binary representation of the low—our compressed binary bit stream

Example:

If the message is "ABCD" and we use these CDFs:

- CDF('A') = 0.2
- CDF('B') = 0.5
- CDF('C') = 0.8
- CDF('D') = 1.0

This means:

- A takes up 0.0–0.2 of the range
- B is 0.2–0.5
- C is 0.5–0.8
- D is 0.8–1.0

Zoom in on the right sub-interval:

- For each character, shrink the current range to the sub-range that corresponds to that character.

- This is done by calculating:

  New range = high - low

  Then:

  high = low + new_range × CDF (current character) low = low + new_range × CDF (previous character)

- Repeat for all characters:

  Each time this operation is carried out, the range [low, high) gets smaller and more precise**,** homing in on the final encoded value.

- Use the final low value**:**

  After cycling through all characters, the final low is somewhere inside a tiny interval that uniquely represents the message**.**

  Convert this low value into a binary number, which becomes the encoded output.

Here, probabilities are used to "zoom in" on a part of the [0,1) interval. Each message lands in its own unique spot in the interval — no overlaps. The smaller the final range, the more precisely it can be represented with a binary fraction. The code successively zooms in on the correct portion of the number line as it processes each character.

Heraclitus Fragment (all upper case; no word spaces or punctuation – in keeping with ancient Greek orthography):

THELORDWHOSEORACLEISATDELPHINEITHERSPEAKSNORCONCEALSBUTGIVESASIGN

Final interval: [0.91405678848989846408, 0.91405678848989846408)

Binary encoding of low value:
11101001111111110100000000101101010000000011101100110000000000000110101000000000111001
1110000000000001101010000111101001011000000011110100101100000001111010010110000 00

Character Probabilities and CDF:
A: P=0.0769, CDF=(0.00000, 0.07692)
B: P=0.0154, CDF=(0.07692, 0.09231)
C: P=0.0462, CDF=(0.09231, 0.13846)
D: P=0.0308, CDF=(0.13846, 0.16923)
E: P=0.1385, CDF=(0.16923, 0.30769)
G: P=0.0308, CDF=(0.30769, 0.33846)
H: P=0.0615, CDF=(0.33846, 0.40000)
I: P=0.0769, CDF=(0.40000, 0.47692)
K: P=0.0154, CDF=(0.47692, 0.49231)
L: P=0.0615, CDF=(0.49231, 0.55385)
N: P=0.0615, CDF=(0.55385, 0.61538)
O: P=0.0769, CDF=(0.61538, 0.69231)
P: P=0.0308, CDF=(0.69231, 0.72308)
R: P=0.0615, CDF=(0.72308, 0.78462)
S: P=0.1077, CDF=(0.78462, 0.89231)
T: P=0.0615, CDF=(0.89231, 0.95385)

U: P=0.0154, CDF=(0.95385, 0.96923)
V: P=0.0154, CDF=(0.96923, 0.98462)
W: P=0.0154, CDF=(0.98462, 1.00000)

Output Bitstream:

- The final compressed bitstream drops from ~259 bits (Huffman) to ~160 bits (Arithmetic), significantly improving the raw compression ratio of this dataset.

12-bit/codon DNA Manifolds

A single 3-base codon only natively encodes 6 bits (4^3=64 states). To get ~10-12 bits per codon, a multi-step "lookup" or "unfolding" is used to access 2^10 to 2^12 states. The DNA Manifold system already provides a recursive approach that can be extended to store more bits per codon.

Dictionary Expansion:

A 2-stage dictionary is used such that codon → dictionary index → final 12-bit value.

Stage 1: The codon's raw 6 bits (C=00, T=01, A=10, G=11 for each base) provides an entry from 0-63.

Stage 2: That entry includes an additional 6-bit (or more) sub lookup that merges in "sub manifold" data.

Net effect: 6 bits from a raw codon + up to 6 bits from a sub-manifold pointer = 12 bits total.

Meta-pointer Overhead:

The original DNA Manifolds approach used 2 or more recursion bits (Amino Code) + 2 "silent" bits (Silent Code). Now these can be stored in the sub-lookup.

Each dictionary entry can embed "meta" data:

- 2 bits for error detection ("Silent Code")
- 2 bits for Manifold recursion pointer
- 2 bits for partial hashing or indexing

[Plus 0-6 bits of actual payload data]

The sub-lookup merges these overhead bits with the codon's original 6 raw bits yielding 10-12 total bits per "unfolded" codon.

Practical Implementation:

- Build a 64-entry "Stage-1 dictionary," each entry referencing up to 16 sub-entries in "Stage-2."

- The sub-manifold is embedded in the same DNA or external table, so each dictionary index "unfolds" to a final 10–12-bit code.

- Carefully avoid codons that create forbidden sequences or repeats (homopolymers, hairpins, restriction sites, etc.).
- The recursion pointer ensures that after reading a codon's stage-1 dictionary index, the decoder knows which sub-dictionary to use.

PARTITION INTO 12-BIT BLOCKS

Why 12 Bits per Codon?

Each 3-base codon natively represents 6 bits (4^3 = 64 states). By using a two-stage dictionary expansion 12 bits can be effectively stored in each codon.

12-Bit Chunking

- Label the compressed stream as $B_1B_2\ldots Bn$ (total bits)
- Break it into chunks C_1, C_2, …, each containing exactly 12 bits (the last chunk might have some 0-padding if needed).

TWO-STAGE MANIFOLD CODING (THE CORE IDEA)

Encoding process keeps with the notion of manifold "pages" except that each codon can store up to 12 bits by combining:

- Stage-1 (6 bits): mapped directly to the 3-base codon
- Stage-2 (6 bits): retrieved from a small sub-dictionary associated with that codon

Thus each codon = Stage-1 index (6 bits) + subindex (6 bits) → total 12 bits stored

Stage-1 (6 bits → 3 bases)

- C = 00, T = 01, A = 10, G = 11 for each base

  Example:

  000000 → CCC
  000001 → CCT
  111111 → GGG

- Skip codons that cause restriction sites or have homopolymers
- Maintain a "safe codon" table for all 64 input patterns

Stage-2 (sub-dictionary for recursion / silent bits)

- Each of the 64 stage-1 indices has up to 64 subentries.
- Each subentry = 6 bits. Here typically split:
    - 2 bits for manifold recursion pointers (which page to decode next or simple "continue vs. jump").
    - 1–2 bits for "silent" or parity/error detection.
- The remainder is actual message payload.
- Since the objective is maximum density, only 2 total bits overhead (1 recursion bit + 1 silent bit) might be used, **leaving 4 bits for payload in the sub-dictionary**, plus 6 bits from stage-1 = 10 bits total. But density can be pushed further. **Overhead can be encoded concurrently** with the 6 bits from stage-1, netting nearly 12 bits.

ENCODE EACH 12-BIT CHUNK → 1 CODON

Below is a step-by-step example for 2 or 3 of the codon mappings to illustrate:

Let C_1 _= "001011 110101" (12 bits as a single binary string)

Stage-1 bits = "001011"$_2$ _= decimal 11
Stage-2 bits = "110101"$_2$ _= decimal 53

→ Stage-1 index 11: Suppose that maps to "CAG" (example only).

→ Then in the sub-dictionary at row #11, subIndex=53 might yield recursion="10", silent="0", leftover payload="101" (some arrangement).

That codon is fully realized as "CAG." The sub-dictionary for #11 is where the other 6 bits are stashed.

Let C_2 _= "111101 001011"

→ Stage-1 bits = "111101"$_2$ _= decimal 61 → "GGT," for example.

→ Stage-2 bits = "001011"$_2$ _= decimal 11 → sub-dictionary row #61, subindex = 11. That might yield recursion = "01", silent = "1", leftover = "011."

This process is repeated for all 12-bit blocks.

ILLUSTRATIVE PARTIAL ENCODING TABLE

A snippet of the Stage-1 dictionary (6 bits → codon) purely as an example:

Index (Dec) | Index (Bin) | 3-Base Codon

0	000000	CCC
1	000001	CCT
2	000010	CCA
3	000011	CCG
	… (middle range) …	
11	001011	CAG
…		

Then, for each row, there is a 64-element sub-dictionary, e.g., row #11 (for “CAG”) might have:

SubIndex	6 bits	recursion bits	silent bit(s)	data bits
0	000000	00	0	000
1	000001	00	0	001
…				
11	110101	11	0	101

(These details can vary to fit overhead budget.)

<u>BINARY STRING HOLDING HERACLITUS FRAGMENT + 52 BITS “METADATA”</u>

111010011111111110100000001011010100000000111011001100000000000001101010000000111001
1110000000000001101010001111010010110000001111010010110000001111010010110000 00

<u>Note that two zeros are added at the end to make up an arbitrary 3rd base in the final codon</u>

<u>RESULTING DNA SEQUENCE</u>

Conversion of all 12-bit chunks → (n) codons → (n) bases yields a final “genestring.” The final output is a single continuous DNA string (plus optional flanking primers or some “stop” codon)

<u>DECODING (MANIFOLD “UNFOLDING”)</u>

- Split the final DNA into 3-base codons.

- For each codon:

  a) Convert base-triplet → 6 bits (stage-1 index).
  b) Look up the corresponding row of your sub-dictionary.
  c) Use the next 6 bits (subIndex) to retrieve recursion bits, silent bits, and the payload.

- Concatenate the recovered 12-bit lumps in correct “page” or linear order.

- Reassemble the arithmetic bitstream.

- Run the arithmetic decoder to recover the exact original text.

EXPECTED BIT DENSITY

Bits per Base

~160 bits stored (including 52 bits of “metadata”)
~41 bases (encoding Heraclitus fragment + “metadata”) $\rightarrow$ 3.902 bits per base

Overhead might push that down slightly (3.6-3.8 bits/base) if more silent bits are added say, for error correction, or certain risky codons are discarded. But achieving 3.5 bits/base is quite feasible with careful design.

PRACTICAL NOTES

- Final chosen “safe codon” table must exclude runs of 4 identical bases and known problematic splice or restriction sites.

- This might reduce the number of codons available below 64 or force some dictionary re-mapping.

- Incorporation of extra parity bits or a short Reed-Solomon code for robust error correction might reduce net density from ~4 bits/base down to ~3.5 bits/base.

- Real synthesis will typically want an additional 20-30 bases of flanking sequence for primers and index barcodes.

CONCLUSION

By:

- Compressing the entire English phrase with an advanced entropy coder (arithmetic coding).

- Splitting the binary output into ~12-bit chunks.

- Using a two-stage manifold design to pack each 12-bit chunk into a single 3-base codon (Stage-1 + sub-dictionary).

- Reserving minimal overhead for recursion/silent bits (1–3 bits).

achievable final encoding realistically approaches or slightly exceeds 3.9 bits/base. The complete “page” recursion concept is preserved, each codon providing a pointer or “silent bits” as desired, yet the overall data density is about as high as currently possible with standard 4-base DNA and robust mapping constraints. This thoroughly encodes English texts into short, highly efficient synthetic DNA sequences.