

THE LOST ALGORITHMS

Volume Five — Compression

By

MARTIN CARR

INDEPENDENT PUBLISHER

Copyright © 2026 by Martin Carr.

All rights reserved.

No part of this book may be reproduced, digitalized, or transmitted in any form without written permission from the publisher, except in the case of brief quotations embodied in critical reviews.

Publisher: Independent Publisher

ISBN Placeholder

TABLE OF CONTENTS

Preface: The Art of Saying the Same Thing in Less	4
How to Read This Book	7
A Note on Patents and the Public Domain	9
Part I — Lossless: Squeezing Without Losing a Bit	10
Runs of the Same Thing (Run-Length Encoding)	11
Short Codes for Common Things (Huffman Coding)	16
Coding by Probability (Arithmetic Coding)	21
Part II — Transforms and Codebooks: Compressing Structure	27
Waves That Undo Themselves (The Reversible Wavelet Transform)	28
A Codebook of Vectors (Vector Quantisation)	34
Folding a 3-D Model (Mesh Compression)	39
Appendix — The Patents of This Volume	45
A Closing Note on the Companion Code	48
Where the Series Goes Next	49

PREFACE: THE ART OF SAYING THE SAME THING IN LESS

Compression is one of the quiet miracles of computing. A photograph that would fill a floppy disk becomes a file you can text to a friend. A film becomes a stream that fits down a phone line. A backup of your entire life's documents fits on a chip smaller than your fingernail. None of this is magic and none of it is free — it is algorithms, a specific body of clever ideas about how to say the same thing in fewer bits, and this book is their history.

The central insight is that most data is redundant — it repeats, it is predictable, it wastes space. A page of text uses the letter e far more than the letter z, yet a naïve encoding spends the same eight bits on each. A photograph's neighbouring pixels are almost always similar. A recording of speech is mostly smooth, occasionally sharp. Compression is the art of noticing that redundancy and spending fewer bits on the common and predictable, more only on the rare and

surprising. Its deepest form, lossless compression, throws nothing away — the original comes back bit-for-bit. Its bolder form, lossy compression, throws away what the eye and ear will not miss, and it is why a photo or a song fits at all.

Like the rest of this series, compression has a patent history, and it is a particularly dramatic one. Some of the field's best ideas were patent-encumbered for decades, which genuinely shaped the technology you use — the world spent thirty years using a worse coding method in JPEG images because the better one was locked behind patents. When those patents expired, the better methods flooded into modern formats. This book tells that story through the algorithms themselves: the run-length coder, the prefix code, the probability coder that was too good to be free, the reversible wavelet at the heart of modern image compression, the codebook that squeezes vectors, and the delta coder that folds a three-dimensional model down to a stream. Each is anchored to a patent that has expired into the public domain, and each is handed to you as code you can run and watch shrink real data.

The honesty this series keeps matters here too. Huffman coding is David Huffman's 1952 algorithm, public domain from birth. Run-length encoding is older than computers. Vector quantisation and wavelets are named academic inventions. Where a chapter cites a specific patent, the Story is clear about what that filing actually claimed and who first had the underlying idea. This is history with

receipts, not a myth of lone invention.

And every algorithm runs on nothing but Python's standard library — no zlib, no Pillow, no numpy. You will watch text shrink to half its size, see a smooth signal collapse into a handful of numbers, and reconstruct it all perfectly, in code small enough to run on a five-pound board. Compression is where information theory meets the file on your disk, and there is no better way to understand it than to make something smaller yourself and then make it whole again.

HOW TO READ THIS BOOK

No mathematics beyond high-school algebra, and every idea is built from zero — bits, probability, and the notion of "information" are all explained as they arrive. Install the companion library:

```
pip install patentml
```

Every chapter names the exact module it draws from, and every number printed in these pages — every compression ratio, every "lossless: True" — came from a real run of that module's demo. Run it and you will see the same file shrink by the same amount.

Read in order or dip in anywhere. The book has two halves. Part I is lossless coding — the entropy methods that shrink data without losing a single bit, and the theory of why there is a hard limit to how far you can go. Part II is structural compression — the transforms

and codebooks that reshape data (images, vectors, 3-D models) so that the entropy coders of Part I can finish the job, and that throw away the imperceptible when perfection is not needed.

A NOTE ON PATENTS AND THE PUBLIC DOMAIN

A patent trades a twenty-year monopoly for the full public disclosure of how an invention works; when it expires, the knowledge belongs to everyone. Compression's history shows both halves of that bargain vividly — ideas locked away for two decades, and then released to reshape the formats we all use. Every patent in this book has reached the end of its term. The implementations here are the author's own MIT-licensed code; the ideas are everyone's. This is a map, not legal advice: confirm the status of any specific patent yourself before building a product on it.

Now: what is the very simplest way to make a file smaller? Notice when it repeats itself.

PART I — LOSSLESS: SQUEEZING WITHOUT LOSING A BIT

The purest form of compression throws nothing away: the original comes back bit-for-bit. This is lossless coding, and it rests on a single idea from information theory — that the number of bits a message truly needs depends on how predictable it is. Common, expected things should cost few bits; rare, surprising things may cost more; and no scheme can beat the "entropy" of the data, a hard floor set by its own statistics. This part climbs the ladder of lossless coding: from the simplest trick of collapsing repeats, to the classic method of giving common symbols short codes, to the near-perfect method that codes an entire message as a single number — the one that was too good to be free.

CHAPTER 1

RUNS OF THE SAME THING (RUN-LENGTH ENCODING)

THE FILING

US Patent 5,748,904 — "Method and system for segment encoded graphic data compression."

Filed 1996. Expired; public domain.

The simplest compression idea of all is to notice when data repeats itself and, instead of writing the same thing over and over, write it once with a count. This patent applies exactly that to graphic data — the long runs of identical pixels that fill the flat regions of an image. Its claims describe encoding each maximal run of an identical value as a (value, length) pair, so that a stretch of a thousand white pixels costs almost nothing. It is run-length encoding, the oldest compression primitive, and the right place to begin because it makes the whole idea of compression concrete in a single line.

THE IDEA

Look at a scan-line across the top of a document image: white, white, white, ... , white, black, black, white, white, Storing each pixel individually spends a byte on every one, even though the line is almost entirely repetition. Run-length encoding (RLE) replaces each run — a maximal stretch of the same value — with a single pair: the value, and how many times it repeats. `aaaabbbbbbbccd` becomes `(a,4)(b,6)(c,2)(d,1)`. Thirteen symbols become four pairs; a thousand identical pixels become one pair. Decoding just expands each pair back into its run.

That is the entire algorithm, and its virtue and its limit are both obvious. On data with long runs — flat regions of an image, whitespace in a document, silence in audio, the same byte repeated — RLE is spectacular, collapsing enormous stretches to almost nothing. On data with no runs — text, noise, already-compressed data where every symbol differs from its neighbour — it does nothing, or even makes things slightly larger, because each single symbol becomes a pair. RLE is a specialist: unbeatable on the redundancy it targets, useless on data without it. That is why it rarely works alone and almost always as a stage — flatten the runs first, then let a smarter coder handle the rest, a pattern that runs through this whole book.

THE STORY

Run-length encoding is older than the patents that apply it and older, arguably, than digital computing — it is the natural first thought anyone has about compression, and it was used in the earliest fax machines and image formats. US5748904's specific setting is graphic data, where RLE shines: document images, faxes, and simple graphics are full of long identical runs, and encoding those runs is enormously effective. Its descendants are everywhere flat data lives. The fax standards that sent documents down phone lines for decades are built on run-length coding of the black-and-white runs across each line. The venerable image formats — BMP, PCX, TGA, and the RLE inside TIFF — use it. It is the first stage of many larger schemes: JPEG run-length-codes the long strings of zeros that its transform produces, and PDF and PostScript include RLE filters. Even the "packing" inside modern formats often begins by collapsing runs. It is the humble foundation the fancier methods stand on.

THE CODE

Walk the data, extending a run or starting a new one; decode by expanding each pair. Here is patentml's `segment_rle`:

```
def encode(data):
    out = []
    for x in data:
        if out and out[-1][0] == x:
```

```
        out[-1] = (x, out[-1][1] + 1)    # extend the current run
    else:
        out.append((x, 1))               # start a new run
    return out

def decode(runs):
    out = []
    for val, count in runs:
        out.extend([val] * count)        # expand each run back
    return out
```

The demo encodes a short run-heavy string and reports the ratio. It prints:

Segment run-length coding (from US5748904, expired):

```
data : aaaabbbbbbbccd
runs : [('a', 4), ('b', 6), ('c', 2), ('d', 1)]
decode: aaaabbbbbbbccd
ratio : 1.62 x
```

The thirteen-character string collapsed to four (value, count) pairs — the four runs it contains — and decoding those pairs reproduced the original exactly (decode equals data, lossless). The compression ratio of 1.62× reflects the modest runs in this short example; on a real document image with runs of hundreds of identical pixels it would

be tenfold or more. The whole method is visible in the runs line: the data's repetition, named once and counted. The module fuzz-tests ten thousand sequences, confirming that encode-then-decode reproduces the original exactly and that runs are always maximal, and reports PASS.

THE INHERITANCE

Run-length encoding is public domain, and it is the tool the instant your data has runs: flat regions in images, repeated bytes, whitespace, silence, sparse data that is mostly one value. It is trivial to implement, impossible to get wrong, and often the single most effective thing you can do to the right data. Its deeper lesson is compositional, and it sets up the rest of the book: RLE rarely works alone, but it is a superb first stage — collapse the obvious repetition, then hand what remains to a smarter coder. Nearly every serious compression format has an RLE step buried in it for exactly this reason. Understanding it is understanding the simplest way redundancy becomes savings, and the pattern of stacking simple stages into powerful pipelines that every later chapter builds on. That first, foundational trick is now yours.

CHAPTER 2

SHORT CODES FOR COMMON THINGS (HUFFMAN CODING)

THE FILING

US Patent 4,698,672 — "Coding system for reducing redundancy."
Filed 1987. Expired; public domain.

Run-length encoding shrinks repetition. This chapter shrinks imbalance — the fact that in almost any data, some symbols are far more common than others, yet a naïve encoding spends the same number of bits on each. This patent concerns a coding system for reducing exactly that redundancy: give the common symbols short codes and the rare symbols long ones, so the average message shrinks. The technique is Huffman coding, and it is not merely good — it is provably optimal among a whole class of codes, one of the most satisfying results in all of computer science.

THE IDEA

Ordinary text spends eight bits on every character, whether it is a space (which appears constantly) or a z (which barely appears at all). That is obviously wasteful: if e is a hundred times more common than z, it deserves a shorter code. Huffman coding builds exactly the code that minimises the average length, given the symbol frequencies.

It works by a beautiful greedy construction. Count how often each symbol appears. Then repeatedly take the two least-frequent symbols (or groups) and merge them into a single node whose frequency is their sum, building a binary tree from the leaves up. The rarest symbols, merged earliest, end up deepest in the tree; the commonest, merged last, end up shallow. Now read each symbol's code off the tree — left is 0, right is 1, from the root down to the symbol's leaf. Common symbols, being shallow, get short codes; rare symbols, being deep, get long ones. Because the codes are the leaves of a tree, no code is a prefix of another (a prefix code), which means a stream of them decodes unambiguously with no separators — you follow the bits down the tree until you hit a leaf, emit that symbol, and start again.

The astonishing part is that this simple greedy procedure is optimal: no other code that assigns a whole number of bits to each symbol can do better on average. And the average length it achieves comes

remarkably close to the entropy of the data — the information-theoretic floor, the true number of bits the data fundamentally requires, below which no lossless scheme can go. Huffman coding gets within one bit of that floor, every time, provably. It is the point where clever engineering meets a hard mathematical limit and very nearly touches it.

THE STORY

The honesty this chapter owes is a lovely piece of history. Huffman coding was invented in 1952 by David A. Huffman, then a graduate student at MIT, who devised it for a term-paper assignment — and famously found the optimal solution that his professor, Robert Fano, had not. It has been public-domain foundational computer science ever since, taught in every algorithms course, never owned by anyone. What US4698672, filed in 1987, and patents like it cover are specific coding systems that apply and refine redundancy-reduction coding in hardware and particular applications — the industrial machinery of using such codes, not the 1952 idea itself. And that idea is one of the most widely deployed algorithms in the world. Huffman coding is the final entropy-coding stage of JPEG images, of the DEFLATE algorithm inside ZIP, gzip, and PNG, of MP3 audio, and of countless other formats. Nearly every file on your computer that has been compressed has passed through a Huffman coder on its way to disk. A student's term paper became one of the load-bearing algorithms of the digital world.

THE CODE

Build the tree by merging the two rarest nodes; read codes off the tree. The demo Huffman-codes a real block of text and measures it against the entropy floor. It prints:

Huffman coding (from US4698672, expired):

```
input      : 291 chars = 2328 bits (fixed 8-bit)
huffman    : 1287 bits (55.3% of original)
avg length : 4.42 bits/symbol
entropy    : 4.37 bits/symbol (Huffman is within 1 bit of this)
lossless   : True
```

The result is the whole theory made concrete. A 291-character text that costs 2328 bits in fixed 8-bit encoding compressed to 1287 bits — 55.3% of the original, nearly halving it, purely by giving common letters short codes and rare ones long codes. The average code length came out at 4.42 bits per symbol — and look at the next line: the entropy of this text, its theoretical minimum, is 4.37 bits per symbol. Huffman landed within five hundredths of a bit of the information-theoretic floor, exactly as the "within one bit of entropy" guarantee promises. And it is lossless — the decoded text is bit-for-bit the original. The module verifies the round-trip, the prefix-code property, and the near-entropy average, reporting PASS.

THE INHERITANCE

Huffman coding is public domain — David Huffman's 1952 gift has always been — and it remains the workhorse entropy coder, reached for whenever data has an uneven symbol distribution, which is nearly always. It is the final squeeze in a huge fraction of the world's compression formats, the stage that turns "we have reduced this to a stream of symbols" into "and here it is in the fewest whole bits per symbol." Its deeper gift is conceptual: it introduces entropy, the hard floor on lossless compression, and shows that a simple greedy algorithm can very nearly reach a fundamental mathematical limit. Understanding Huffman is understanding why compression has limits at all, and how close clever coding can come to them. The one gap it leaves — that it must spend a whole number of bits per symbol, and so falls just short of entropy — is precisely the gap the next chapter closes, with a method so good it was kept from the public for decades. Huffman's optimal prefix code is now yours.

CHAPTER 3

CODING BY PROBABILITY (ARITHMETIC CODING)

THE FILING

US Patent 6,424,956 — "Stochastic encoder/decoder/predictor."
Filed 1999. Expired; public domain.

Huffman coding is optimal — but only among codes that spend a whole number of bits per symbol. That constraint leaves a little on the table: a symbol with probability that would ideally cost 2.3 bits must be given 2 or 3, never 2.3. This chapter's method escapes that constraint entirely, coding an entire message as a single number to fractional-bit precision, and reaching the entropy floor almost exactly. It is arithmetic coding, and its story is the most dramatic in this book: it was better than Huffman, and for that very reason it was locked behind patents for a generation.

THE IDEA

Huffman assigns a code to each symbol. Arithmetic coding refuses to. Instead it codes the whole message at once, as a single fractional number in the interval between 0 and 1, narrowing that interval symbol by symbol according to each symbol's probability.

Picture the interval $[0, 1)$ divided into slices, one per symbol, each slice as wide as that symbol's probability — a common symbol gets a fat slice, a rare one a sliver. To encode the first symbol, zoom into its slice: that becomes your new working interval. Now subdivide that interval by the same probabilities and zoom into the second symbol's sub-slice. Each symbol narrows the interval, more for rare symbols (which cut it down a lot) and less for common ones (which barely shrink it). After the whole message, you are left with a tiny final interval, and you output any single number inside it. To decode, you reverse the zoom: see which symbol's slice your number falls in, emit it, rescale, and repeat.

The magic is that a probable message ends up in a wide final interval, which can be pinpointed with few bits, while an improbable message ends up in a narrow one needing more — and this happens continuously, to fractional-bit precision. The symbol that ideally costs 2.3 bits genuinely costs about 2.3 bits, because the interval-narrowing does not round to whole bits the way a per-symbol code must. Arithmetic coding therefore reaches the

entropy floor almost exactly, beating Huffman's up-to-one-bit-per-symbol overhead — an overhead that matters enormously when symbols are highly skewed (a symbol with probability 0.9 ideally costs 0.15 bits, but Huffman must spend a whole one, a near-sevenfold waste that arithmetic coding avoids).

THE STORY

This is the famous one. Arithmetic coding was developed in the 1970s, principally by Jorma Rissanen and Richard Pasco, and refined at IBM by Glen Langdon and others — a genuine advance over Huffman, reaching entropy where Huffman only approached it. And then it ran headlong into patents. IBM and others held a thicket of patents on arithmetic coding and its efficient variants (the "Q-coder" and its kin), and those patents genuinely shaped the technology: the original JPEG standard specified an arithmetic-coding mode that was better than its Huffman mode — and almost nobody used it, for decades, because it was patent-encumbered. The world compressed billions of images with the inferior coder to stay clear of the patents. It is one of the clearest cases in computing of intellectual property holding back deployment of a superior technique. The patents, including filings like US6424956, eventually expired — and arithmetic coding and its close relative range coding flooded into modern formats the moment they were free. Today they are everywhere the best compression is demanded: CABAC, the arithmetic coder in H.264 and H.265 video (the codecs behind most

streaming and broadcast), the entropy stage of JPEG 2000, the coders inside modern archivers and the state-of-the-art compressors. The method that was too good to be free is now the default at the cutting edge.

THE CODE

Narrow the interval by each symbol's probability slice; output a number inside the final interval; reverse to decode. Here is the encoder from `patentml`'s `arithmetic_coder` (exact rational arithmetic, so it is precise and clear):

```
def encode(message, model):
    table, _ = _cum(model)
    lo, hi = Fraction(0), Fraction(1)
    for sym in message:
        s_lo, s_hi = table[sym]
        span = hi - lo
        hi = lo + span * s_hi      # zoom into this symbol's slice
        lo = lo + span * s_lo
    return (lo + hi) / 2          # any point in the final interval
decodes the same
```

The demo codes a message under a skewed model and round-trips it. It prints:

Arithmetic coder (from US6424956, expired):

```
model    : {'a': 5, 'b': 3, 'c': 1, 'd': 1}
message  : aaabac
code     : 0.0784375000
decoded  : aaabac
```

The entire six-symbol message aaabac was encoded as the single number 0.0784375 — not a sequence of per-symbol codes, but one fraction that pins down the whole message. Under the model, a is the most probable symbol (frequency 5 of 10), so each a barely narrowed the interval and cost very little; the rarer b and c narrowed it more and cost more — coding by probability, to fractional-bit precision. Decoding the number 0.0784375 walked back through the same interval slices and recovered aaabac exactly. The module fuzz-tests three thousand random messages and models, confirming that encode-then-decode round-trips exactly, and reports PASS.

THE INHERITANCE

Arithmetic coding is public domain at last, and it is the entropy coder to reach for when you want the best lossless compression a symbol model can give — especially when symbols are highly skewed, where its fractional-bit precision decisively beats Huffman's whole-bit rounding. Paired with a good probability model (even a simple adaptive one that updates frequencies as it goes), it is the

finishing stage of the most powerful modern compressors and video codecs. But its greatest lesson is the cautionary tale in its Story: here is a superior algorithm whose deployment was held back for decades by patents, so that the world used a worse method in its most common image format until the monopolies lapsed. It is the sharpest illustration in this entire series of what the patent bargain costs while it runs and gives back when it ends — the knowledge was published, disclosed in full, and yet locked from practical use for twenty years. Now it is free, and it is yours. That is the whole reason this series exists.

PART II — TRANSFORMS AND CODEBOOKS: COMPRESSING STRUCTURE

The lossless coders of Part I shrink data by exploiting the statistics of its symbols. But the biggest wins in compressing images, sound, and 3-D models come earlier — from reshaping the data so that its redundancy becomes obvious to those coders, and from throwing away what the eye and ear will never miss. This part is about that reshaping. A wavelet transform gathers a signal's energy into a few large numbers and a sea of near-zeros. A vector codebook replaces whole blocks of data with references to a small dictionary of representatives. A delta coder folds a three-dimensional model down to a stream of small differences. Each prepares structured data so the entropy coders of Part I can finish the job — and each, when perfection is not required, opens the door to the far larger savings of lossy compression.

CHAPTER 4

WAVES THAT UNDO THEMSELVES (THE REVERSIBLE WAVELET TRANSFORM)

THE FILING

US Patent 5,867,602 — "Reversible wavelet transform and embedded codestream manipulation."

Filed 1994. Expired; public domain.

An entropy coder can only exploit redundancy it can see. Raw image pixels hide their redundancy — neighbouring pixels are similar, but that similarity is not something a symbol coder can grab. A transform exposes it, rewriting the image in terms of smooth trends and sharp details so that the smooth parts (nearly everything) collapse toward zero. This patent describes a reversible wavelet transform — one that reshapes an image losslessly, into a form where its energy is concentrated and most coefficients are near zero — and the embedded codestream that made it the heart of JPEG

2000.

THE IDEA

A wavelet transform splits a signal into two halves: a low-pass part capturing its smooth trend (the local averages), and a high-pass part capturing its details (the local differences). Apply it once and you have a half-size smooth approximation plus a set of detail coefficients; apply it again to the approximation and you get a quarter-size smoother approximation plus more details; and so on. The signal is reorganised into a coarse summary and successive layers of fine detail.

The point, for compression, is energy compaction. Most real signals — images especially — are mostly smooth, so almost all their "energy" gathers into the low-pass approximation, and the detail coefficients become tiny or exactly zero. A smooth region that filled a thousand pixels becomes a handful of large approximation numbers and a thousand near-zeros — and near-zeros are exactly what the entropy coders of Part I devour. Run-length code the zeros, Huffman or arithmetic code the rest, and the image shrinks dramatically, all because the transform moved the redundancy into plain view.

The word reversible is the patent's key contribution. Ordinary wavelet transforms use fractional arithmetic and lose a little

precision, making them lossy. The reversible integer wavelet — the famous 5/3 transform — uses only integer additions and shifts arranged (via a technique called lifting) so that the exact original comes back, bit-for-bit. This is what lets a single transform serve both masters: keep all the coefficients and you have lossless compression; drop or coarsen the smallest ones and you have lossy compression at whatever quality you choose — from the same reshaping of the same data.

THE STORY

Wavelets are a deep and beautiful piece of twentieth-century mathematics, developed through the 1980s by Ingrid Daubechies, Stéphane Mallat, Yves Meyer and others — a way to analyse signals at multiple scales at once, foundational and widely published. The specific reversible integer wavelet (the Cohen–Daubechies–Feauveau 5/3 filter) and the "embedded codestream" idea — where the compressed data is ordered so you can stop reading at any point and get a lower-quality version, enabling progressive and scalable images — are what patents like US5867602 industrialised in the mid-1990s. Their great destination was JPEG 2000, the standard built to succeed JPEG: its lossless mode is exactly this reversible 5/3 wavelet, and its famous quality-scalability (one file that can be decoded at many resolutions and qualities) is the embedded codestream. JPEG 2000 runs in medical imaging, digital cinema (it is the mastering format for

movies), satellite and archival imagery, and more. Wavelets also power the FBI's fingerprint compression and a generation of image and signal analysis. A transform that gathers a signal's energy into a few numbers turned out to be one of the most useful ideas in all of signal processing.

THE CODE

The 5/3 lifting step predicts details from neighbours and updates the approximation, all in integers so it inverts exactly. The demo transforms a signal and an image and checks losslessness and compaction. It prints:

Reversible wavelet transform (from US5867602, JPEG2000 5/3 core):

```
signal : [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8]
approx L2: [2, 6, 5]
details : [[-2, -3, 6, 3, -2, 3], [-1, -1, 0]]
inverse : [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8]
lossless : True
```

smooth ramp(64): 31/32 detail coeffs are 0 (energy compacted into approx)

8x8 smooth image: 40/48 high-freq coeffs are 0 (energy in LL)

2D lossless (idwt2==img): True

Both properties the compression depends on are visible. First, losslessness: the transformed signal, inverted, returns the exact original (inverse equals signal, bit-for-bit), and the same holds for the 2-D image — the integer lifting undoes itself perfectly. Second, and this is where the compression comes from, energy compaction: on a smooth 64-sample ramp, 31 of 32 detail coefficients came out as exactly zero — all the information collapsed into the tiny approximation — and on a smooth 8×8 image, 40 of 48 high-frequency coefficients are zero. The transform turned a signal into a few meaningful numbers and a sea of zeros, precisely the form an entropy coder crushes. The module verifies 1-D and 2-D lossless round-trips and the compaction, reporting PASS.

THE INHERITANCE

The reversible wavelet transform is public domain, and it is the tool for compressing anything smooth and multi-scale — images above all, but also audio, scientific and sensor data, any signal whose energy concentrates at coarse scales. Its double gift is rare and valuable: the same transform gives you lossless compression (keep everything) or lossy compression at any quality (discard the smallest details), because it reshapes the data without committing to either. Paired with the entropy coders of Part I, it is a complete modern image compressor in spirit — reshape with the wavelet, then code the zeros and the survivors. Understanding it demystifies JPEG 2000, wavelet-based denoising, multi-resolution imagery, and the

whole idea that the smartest thing to do with structured data is often not to code it directly but to transform it into a shape where its redundancy is impossible to miss. That transform, and a working reversible wavelet, are yours now that this filing has expired.

CHAPTER 5

A CODEBOOK OF VECTORS (VECTOR QUANTISATION)

THE FILING

US Patent 6,421,467 — vector-quantisation-based compression (Texas Tech University).

Filed 1999. Expired 2019; public domain.

The coders so far compressed one symbol, or one number, at a time. But data often comes in blocks — a patch of an image, a frame of audio, a chunk of a feature vector — and blocks have structure across their whole extent that per-symbol coding misses. This chapter compresses whole blocks at once by building a small dictionary of representative blocks and replacing each real block with a reference to its nearest dictionary entry. It is vector quantisation, the idea of compressing by approximating with a codebook, and it reaches from 1980s speech coding all the way into

the vector databases powering today's AI.

THE IDEA

Round a number to the nearest whole value and you have scalar quantisation — crude compression, since you now store a short index instead of a precise value, at the cost of a little error. Vector quantisation (VQ) does the same thing but to whole vectors — blocks of numbers — at once, and that "at once" is where its power lives.

Gather a representative sample of the blocks your data contains — image patches, sound frames, feature vectors — and find a small set of codewords, say 256 of them, that best represent the whole population: cluster the sample and take the cluster centres. That set is the codebook. Now, to compress any block, find the codeword nearest to it and store just its index — one byte for a whole block. To decompress, look the index up in the codebook and use that codeword as the (approximate) block. A patch of an image that was 64 numbers becomes a single index into a dictionary of 256 typical patches. The compression is enormous; the cost is that you get back the nearest codeword, not the exact block — VQ is inherently lossy, and its quality is set by how large and well-chosen the codebook is.

The classic way to build the codebook is the LBG algorithm (Linde–Buzo–Gray): start with some codewords, assign every

training block to its nearest one, move each codeword to the average of the blocks assigned to it, and repeat until the total distortion stops falling — essentially k-means clustering in the service of compression. A refinement for high-dimensional data, product quantisation, splits each vector into several sub-vectors and quantises each with its own small codebook, so that a modest number of sub-codewords combine to represent an astronomical number of full vectors cheaply. patentml's module implements all three: the batch LBG, an online adaptive version, and the product quantiser.

THE STORY

Vector quantisation is grounded in the work of Yoshihiro Linde, Andrés Buzo, and Robert Gray, whose 1980 LBG algorithm made codebook design practical, building on Gray's and others' information-theoretic foundations. It became a major technique in speech coding (representing frames of speech with codebook indices, central to the low-bitrate voice codecs of digital telephony) and in image compression. Patents like US6421467 industrialised particular VQ-based compression schemes through the 1990s. But VQ's most surprising second life is happening right now. Product quantisation turned out to be the key to approximate nearest-neighbour search over billions of vectors — the technology behind modern semantic search, recommendation, and the retrieval systems that feed large language models. When an AI system finds the most relevant documents to your query by comparing embedding vectors, it very

likely uses product quantisation to compress those billions of vectors so they fit in memory and can be searched fast. A 1980 speech-compression idea became, forty years later, load-bearing infrastructure for artificial intelligence. Few algorithms have had a stranger or more consequential afterlife.

THE CODE

Build a codebook by clustering; compress by storing nearest-codeword indices. The demo builds three kinds of quantiser and reports their reconstruction distortion. It prints:

LBG codebook (8 codewords, 8-D): distortion=0.2875

AdaptiveVQ codebook: distortion=0.2899

ProductQuantiser: mean reconstruction error on 5 vectors = 0.8178

Each line is a codebook doing its job. The LBG quantiser built 8 codewords in 8-dimensional space and represents the data with a mean distortion of just 0.2875 — every block of 8 numbers reduced to a single 3-bit index (one of 8), reconstructed to within that small error. The AdaptiveVQ, which builds its codebook online as data streams past rather than in a batch, reaches essentially the same quality (0.2899) without ever seeing all the data at once. And the ProductQuantiser splits each vector into sub-vectors and quantises each separately, reconstructing with a modest error (0.8178) — the technique that scales to billions of vectors. Blocks became indices,

indices became approximate blocks, at controllable cost. The module verifies all three quantisers against their reconstruction targets and reports PASS.

THE INHERITANCE

Vector quantisation is public domain, and it is the tool whenever you want to compress blocks of data by approximating them from a dictionary: image and audio compression, of course, but increasingly the compression of the embedding vectors at the heart of modern search and AI. Its trade is explicit and tunable — a bigger codebook means better fidelity and larger indices; a smaller one means more compression and more error — so you dial in exactly the quality you can afford. Its deepest lesson is that compression and clustering are the same act seen from two angles: to compress is to find the few representatives that stand in for the many, which is precisely what clustering does. That unity, and a working set of quantisers — batch, adaptive, and product — are yours now that this filing has expired, a forty-year-old idea busier today than it has ever been.

CHAPTER 6

FOLDING A 3-D MODEL (MESH COMPRESSION)

THE FILING

US Patent 5,870,094 — "System and method for transferring compressed three-dimensional graphics data."

Filed 1995. Expired; public domain.

The book ends with a kind of data the earlier chapters never touched: three-dimensional models. A mesh — the network of vertices and triangles that describes a 3-D object — is bulky, a long list of precise coordinates, and transmitting it (to a game, a browser, a remote renderer) demands compression of its own. This patent's claims describe transferring compressed 3-D graphics data by exploiting the coherence of a mesh: successive vertices are close together, so storing their small differences rather than their absolute positions shrinks the model. It is delta coding applied to geometry — the

compression that streams 3-D worlds.

THE IDEA

A 3-D model is, at heart, two lists: the vertices (points in space, each an x-y-z coordinate) and the faces (triangles, each naming three vertices). Stored naively, every vertex is three full-precision numbers, and a detailed model has millions of them — a great deal of data to store or send.

But meshes are coherent: when you list a model's vertices in a sensible order, each one is near the previous one, because a surface is continuous and you tend to walk along it. That means the difference between consecutive vertices is small, even when the absolute coordinates are large. Delta coding exploits this directly: store the first vertex in full, then store each subsequent vertex as its difference from the one before. A vertex at (1000, 2000, 3000) followed by one at (1001, 2000, 3001) is stored as the deltas (+1, 0, +1) — tiny numbers, cheap to encode, instead of large absolute ones. Decoding just walks the deltas, adding each to a running position to rebuild the exact vertices. The faces, being integer indices, come along unchanged.

The result is a mesh whose bulky coordinate list has become a small starting point plus a stream of tiny differences — and those tiny, mostly-similar differences are, once again, exactly the low-entropy

data that the coders of Part I compress beautifully. Delta coding does not finish the job; it sets up the job, converting large absolute coordinates into small deltas that an entropy coder then crushes. And because the deltas reconstruct the vertices exactly, this stage is perfectly lossless — the model that comes out is the model that went in.

THE STORY

Compressing 3-D geometry became urgent in the mid-1990s as interactive 3-D graphics moved from specialist workstations toward consumer machines and, soon, the network — models needed to be transmitted, not just stored. US5870094, filed in 1995, addresses exactly that: transferring compressed 3-D graphics data. It belongs to the founding era of geometry compression, alongside the influential work of Michael Deering at Sun (whose 1995 "Geometry Compression" paper framed the field) and Gabriel Taubin and others at IBM. The techniques they pioneered — delta-coding vertex positions, cleverly encoding the triangle connectivity, quantising coordinates to the precision the eye needs — are the foundation of every modern 3-D transmission format. Their descendants run the 3-D web and beyond: glTF with its Draco compression (which delta-and-entropy-codes meshes to stream models to browsers and games), the compressed geometry in game engines and streaming worlds, the point-cloud and mesh compression standards for volumetric video and 3-D capture, and the model formats behind AR

and VR. Every time a detailed 3-D object loads quickly over a network, geometry compression in this lineage is folding it down for the trip.

THE CODE

Store the first vertex, then successive differences; decode by accumulating them. Here is the encoder from patentml's `mesh_compress`:

```
def encode(vertices, faces):
    deltas = []
    prev = (0, 0, 0)
    for v in vertices:
        deltas.append((v[0] - prev[0], v[1] - prev[1], v[2] - prev[2])) #
    difference
    prev = v
    return {"n": len(vertices), "deltas": deltas, "faces": [tuple(f) for f in
faces]}
```

The demo compresses a small mesh and confirms an exact round-trip. It prints:

3-D mesh compression (from US5870094, expired):

vertices: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (0, 1, 1)]

deltas : [(0, 0, 0), (1, 0, 0), (0, 1, 0), (-1, 0, 1)]

round-trip exact: True

The transformation is right there in the two lists. The vertices — points scattered in space — became a stream of deltas, each just the small step from the previous vertex: (1,0,0), then (0,1,0), then (-1,0,1). Every delta is a tiny number in {"1, 0, real model the absolute coordinates would be large — and tiny numbers are what entropy coders compress to almost nothing. Decoding accumulated the deltas back into the exact original vertices: the round-trip is exact, perfectly lossless. On a coherent million-vertex model, this turns a flood of large coordinates into a stream of small, similar differences — the setup that makes real mesh compression work. The module fuzz-tests five thousand meshes, confirming that delta encode-then-decode reconstructs the geometry exactly, and reports PASS.

THE INHERITANCE

Mesh compression by delta coding is public domain, and it is the tool for transmitting or storing any coherent geometric data: 3-D models for games and the web, point clouds from scanners and depth cameras, trajectories, and any sequence of positions where each is near the last. Store the differences, not the absolutes, and the data collapses toward the low-entropy form the earlier chapters' coders finish off. It is a fitting end to the book, because it distils

compression's master pattern one last time: do not compress the data directly — reshape it so its redundancy becomes obvious, then let a simple coder do the work. Runs, uneven frequencies, probability, smooth trends, block similarity, spatial coherence — every chapter has been a different way of seeing redundancy that was there all along, and then spending fewer bits on it. That way of seeing, and a shelf of working coders to apply it, are yours now that these patents have expired.

And there the tour ends. From the run-length coder that collapses repetition, through Huffman's optimal codes and the arithmetic coder that was too good to be free, to the wavelets, codebooks, and delta coders that reshape structured data — you have walked the machinery that makes the world's files small enough to store, send, and stream. Every algorithm was published in a patent, monopolised for twenty years, and then set free; here they are as code you can run on a five-pound board, watching real data shrink and come back whole. The vault is open. Go and build.

APPENDIX — THE PATENTS OF THIS VOLUME

Compression's foundations are famous, published, and in several cases never patented at all — so this appendix, like the cryptography volume's, separates the specific patents each chapter's code is anchored to from the foundational work the Story sections credited. Huffman coding was public domain from birth; run-length encoding predates computing; wavelets, vector quantisation, and geometry compression are named academic inventions. Where a chapter cites a patent, it is an industrial application of an older idea, and the book says so. All filings below have reached the end of their term.

PATENTS ANCHORING THE CHAPTERS

Ch	Patent	Title	Filed
----	--------	-------	-------

----	-----	-----	-----
------	-------	-------	-------

1	US5748904	Method and system for segment encoded graphic	
---	-----------	---	--

data compression | 1996 |
| 2 | US4698672 | Coding system for reducing redundancy | 1987 |
| 3 | US6424956 | Stochastic encoder/decoder/predictor | 1999 |
| 4 | US5867602 | Reversible wavelet transform and embedded
codestream manipulation | 1994 |
| 5 | US6421467 | Vector-quantisation-based data compression (Texas
Tech University) | 1999 |
| 6 | US5870094 | System and method for transferring compressed
three-dimensional graphics data | 1995 |

FOUNDATIONS CREDITED IN THE STORY SECTIONS

Idea	Origin	Credited to
-----	-----	-----
Huffman coding	1952, MIT term paper	David A. Huffman — public domain from inception
Run-length encoding	Predates digital computing	No single inventor; earliest fax/image formats
Arithmetic coding	1970s, IBM and academia	Jorma Rissanen, Richard Pasco, Glen Langdon
Wavelet transforms	1980s	Ingrid Daubechies, Stéphane Mallat, Yves Meyer; 5/3 filter: Cohen–Daubechies–Feauveau
Vector quantisation (LBG)	1980	Yoshihiro Linde, Andrés Buzo, Robert Gray
3-D geometry compression	mid-1990s	Michael Deering (Sun), Gabriel Taubin (IBM), and others

The most instructive entry is arithmetic coding (Chapter 3): a technique better than Huffman, published in the 1970s, yet held back from practical use for decades by patents — so that the world's most common image format, JPEG, used the inferior Huffman coder until those patents expired. It is the sharpest example in this whole series of what the patent bargain costs while it runs and returns when it ends.

CHAPTER 7

A CLOSING NOTE ON THE COMPANION CODE

Every algorithm in this book ships in the `patentml` library:

```
pip install patentml
```

Each module names its patent, carries a runnable demo that builds its own test data, and is verified against a reference — the same `PASS` quoted at the end of each chapter's Code section, with nothing outside Python's standard library. Watch text halve, a smooth signal collapse to a few numbers, a mesh fold to a stream of deltas — and reconstruct all of it exactly. The book tells you why and when; the library lets you run, change, and break it.

CHAPTER 8

WHERE THE SERIES GOES NEXT

The Lost Algorithms of Compression is Volume Five. Volumes One and Two told machine learning's patent history; the Computer Vision volume told the story of the pixel; Volume Three told how the world's data got fast; Volume Four told the story of secrecy and trust. One volume remains in the current arc — Sound (how space and speech fit down a wire) — drawn, like all the rest, from patents that expired into the public domain and were picked up and handed over as working code.

A patent is a bargain: a twenty-year monopoly in exchange for publishing exactly how an invention works. Compression's history shows that bargain at its most vivid — brilliant ideas disclosed in full, locked from use for a generation, and then set free to reshape every file we store and stream. The vault is open. Go and build.