MathQR: Canonical, Copy-Safe Math Strings with Standard QR Codes

Douglas C. Youvan

doug@youvan.com

www.youvan.ai

October 6, 2025

Mathematical expressions travel poorly across software: LaTeX is verbose and fragile, Unicode math is compact but not copy-safe everywhere, and screenshots aren't machine-readable. We propose MathString—a canonical, ASCII (or ultra-compact) representation of equations—and MathQR, a standards-compliant pipeline that packs these strings into ordinary QR codes. Our goal isn't just to "stuff text in a QR," which is well-known, but to make the payload mathematically deterministic: different notations normalize to the same bytes; decoding a QR always yields the same canonical string. We achieve this with (i) a tiny math tokenizer/operator dictionary (e.g., Dv, Cr, Dt, L2), (ii) optional macros for heavy sub-expressions, and (iii) an optional metadata shim (IDs, units, checksums). Because MathQR uses standard Model-2 QR with Reed–Solomon error correction, any phone scanner returns the canonical string, while our tools can round-trip to richer renderings (glyphs, metrics) when available. We demonstrate the approach on classic systems—Maxwell, heat, Schrödinger—and show that single equations fit in small versions at high error-correction levels, while multi-equation bundles remain compact. The result is a practical, interoperable way to print, scan, diff, archive, and compose mathematics across platforms without bespoke apps or fonts.

Keywords: canonical math string, QR code, MathQR, UltraString, math tokenization, deterministic encoding, copy-safe math, LaTeX alternative, interoperability, error correction, metadata, macros.

A collaboration with GPT-5. CC4.0.

Motivation

Modern math travels through too many brittle formats. Authors draft in LaTeX, paste Unicode into chats, export PDFs for reviewers, and screenshot equations for slides. Each hop risks ambiguity or loss of structure. We want a path where the *same* math survives copy/paste, printing, scanning, archiving, and machine use—without custom apps or fonts.

Pain points

TeX verbosity.

- LaTeX is expressive but wordy and layout-centric. Small statements turn into long control sequences, making diffs noisy and search brittle.
- Equivalent expressions often have multiple TeX spellings; round-tripping through different editors or exporters can change spacing/macros without changing meaning.
- TeX isn't directly consumable by non-TeX tools; you typically need a renderer or a converter, which complicates automation.

Unicode portability.

- Pretty symbols are compact, but support is uneven across platforms, fonts, and editors.
- Copying from PDFs or slides can corrupt characters (e.g., ∇, $\hbar$, ε) or reorder combining marks.
- Even when it displays, Unicode alone doesn't encode *structure* (e.g., what's an operator vs. an identifier), limiting reliable parsing.

Screenshots.

- Images are visually faithful but not machine-readable.
- OCR on math is brittle, loses semantics, and fails on small print or low contrast.
- Images don't diff, don't search, and don't compress semantically.

Design goals

Canonical.

- Normalize diverse notations to a single, deterministic **MathString**: fixed token set, spacing rules, and operator names (e.g., Dv, Cr, Dt, L2).
- Same math → same string → same bytes. Determinism enables robust hashing, deduplication, and citation.

Compact.

- Keep it short enough for human scanning and storage, with an **UltraString** mode (no spaces, heavy abbreviations) when density matters.
- Support optional **macros** for repeated heavy subexpressions to shrink payloads further.

Round-trippable.

- The canonical string is primary; everything else (glyphs, TeX, MathML) is a derived view.
- Printed artifacts remain recoverable: standard **QR payloads** always decode back to the exact canonical string, optionally with metadata and checksums.

Standard-compliant.

- Use **Model 2 QR**, standard Reed–Solomon ECC, and Byte mode so any off-the-shelf scanner/phone works—no bespoke app required.
- Keep metadata lightweight and backward-compatible; older decoders ignore flags they don't understand, while our tools exploit them.

Together, these goals produce a practical pipeline: write once in a canonical, copy-safe form; share as text or QR; recover exactly the same math everywhere.

MathString & UltraString

Normalization rules (spaces, tokens, binders)

Goal: one mathematical statement → **one** canonical byte string.

Whitespace

- Collapse runs of spaces/tabs to a **single space**.
- **Exactly one** space around top-level infix ops: = + - * / ^ < <= > >= and or -> <->.
- **No** spaces after function names or before parentheses: f(x,y), Lap(u).
- **No** trailing/leading spaces; no spaces just inside delimiters: (a,b), {x|P(x)}, [a,b].

Tokens (fixed names, ASCII)

- Calculus/linear ops: grad (Gd), div (Dv), curl (Cr), Lap (L2), d/dt, d/dx, int_{a}^{b}, lim_{x->a}.
- Linear algebra: diag, trace, transposes A', pseudo-inverse A^+.

- Logic/relations: and or not -> <-> = != < <= > >= in notin subset subseteq.
- Probability: P(A), E[X], Var[X], Cov[X,Y], ~.
- Sets & builders: {x|P(x)}, {a,b,c}, tuples (a,b), intervals (a,b].
- Constants: pi, e, i, hb, e0, m0, c, G, Lam.

Binders (quantifiers & declarations)

- Canonical header order: forall → exists! → exists (then assumptions).
- Each binder in context: as forall x in S, exists y, exists! z.
- Type/space declarations normalize to in with canonical symbol names: x in R^n, A in R^{n x n}.
- Numeric literals: normalize to **scientific or integer** strings (no locale commas): 1e-3, 42, 3/5.

Operator precedence (for stable spacing/parentheses)

1. Function call/indexing f(x), A[i]
2. Power ^
3. Unary -, not
4. Multiplicative *, /
5. Additive +, -
6. Relational = != < <= > >= in ...
7. Logical and or -> <->

- Canonicalizer inserts minimal parentheses to preserve meaning.

Examples (before → MathString)

- ∇×E = −∂B/∂t → curl(E) = - dB/dt
- iℏ ψ_t = - (ℏ^2/2m) Δψ + Vψ → i*hbar*d/dt psi = (-hbar^2/(2*m))*Lap(psi) + V*psi
- ∀x∈ℝ: x²≥0 → forall x in R: x^2 >= 0

UltraString v0 (no-space, heavy abbreviations)

Purpose: maximum density for print/QR; readability comes later.

Rules

- **No spaces** anywhere.
- Multiplication by **juxtaposition**: nuL2(u) means nu * Lap(u).
- Fixed short op codes:
    - Gd=grad, Dv=div, Cr=curl, L2=Lap, Dt=d/dt, Dx=d/dx
    - Logic: & for and, | for or, ->, <->
    - Relations unchanged: = <= >= < > !=
- Constants short names: hb,e0,m0,pi,G,c,Lam.
- Headers optional; if used, context: becomes C: and claim: becomes K:.

Examples

- Maxwell (Ampère–Maxwell): Cr(B)=m0J+m0e0Dt(E)
- Heat: Dt(u)=alphaL2(u)
- Navier–Stokes (incomp.): Dv(u)=0;Dt(u)+u.Gd(u)=-Gd(p)+nuL2(u)+f

(Semicolons are allowed as intra-line separators when needed.)

Abbreviation table (core)

Gd grad Dv div Cr curl L2 Lap Dt d/dt Dx d/dx

hb hbar e0 eps0 m0 mu0 Lam Lambda

Optional macros and metadata; determinism & checksums

Macros (size wins without ambiguity)

- File-local, explicit, single-pass, prefix @:
    - Declaration: @H(psi)=(-hb^2/(2m))L2(psi)+Vpsi
    - Use: ihbDt(psi)=H(psi)

- Macro identifiers are **bound** (no capture): only names you define expand; standard tokens (Gd,Dv,Dt,…) never change.
- During canonicalization to **MathString**, macro **expands** unless the target is **UltraString for QR**, where the macro may remain to save bytes; either way, round-trip is deterministic.

Metadata block (optional, ignored by naïve readers)

- Bracketed pragma after the statement:
    - @[id=Maxwell_SI_Diff_AmpereMaxwell, unit=SI, roles=field,source, sha1=1f3870be...]
- Fields:
    - id (stable name), ns (namespace), unit, tags, roles, trace (e.g., CODATA), notes
    - sha1 is a **content hash** of the canonical MathString *without* metadata.

Determinism (how we guarantee same bytes)

1. **Normalize**: apply all spacing, precedence, token, and binder rules.
2. **Constant folding of forms**: numeric literals go to canonical decimal/scientific; rationals as p/q.
3. **Sort binders & assumptions** in a fixed order; rename dummy indices only when required to avoid clashes (i,j,k order).
4. **Macro resolution mode** recorded in a flag: expand (MathString) or keep (UltraString).
5. **Checksum**: compute sha1(canonical_bytes) and optionally include a truncated prefix (e.g., 7 bytes) in metadata or QR payload; this lets decoders **verify round-trip integrity**.

Mini examples (with metadata)

K: Cr(B)=m0J+m0e0Dt(E) @[id=Maxwell_AmpereMaxwell, ns=EM, sha1=9a1b2c3d]

K: Dt(u)=alphaL2(u) @[id=Heat, ns=PDE, unit=SI]

Round-trip expectations

- **Text → MathString → UltraString → QR → scan → UltraString → MathString** yields **bit-identical** MathString.
- If macros were kept in UltraString, the decoder either **expands** using attached macro table or emits the macro plus its definition; either path still verifies against sha1.

When to use what

- **MathString**: drafts, diffs, reviews, archival text.
- **UltraString**: printing, QR payloads, labels, constrained storage.
- **Metadata**: catalogs, experiments, constants with traceability; harmless to omit for short notes.

This division keeps human-readable math as the source of truth, while giving you a tiny, deterministic form for machines and paper.

MathQR (Encoder)

Pre-encode: tokenize → opcodes → (optional) compress

Purpose: shrink and stabilize the payload *before* handing it to a standard QR encoder.

1. **Tokenize (deterministic)**
    - Split UltraString into tokens from a fixed dictionary: operators (= + - * / ^ ( ) { } [ ] , : ;), math ops (Dv, Cr, Dt, L2, Gd), constants (hb, e0, m0, pi, G, c, Lam), identifiers (E, B, J, u, psi, …), numbers (123, 3.2e-5, p/q).
    - Enforce canonical casing and numeric forms.
2. **Map tokens to opcodes**
    - Reserve single-byte codes (0x00–0x7F) for common tokens. Examples:
        - Cr→0x13, Dt→0x14, Dv→0x15, L2→0x16, Gd→0x17
        - punctuation: =→0x20, +→0x21, -→0x22, (→0x28,)→0x29
        - constants: m0→0x30, e0→0x31, hb→0x32, pi→0x33, G→0x34, c→0x35, Lam→0x36
    - **Identifiers & numbers**: use a short form
        - 0xF0 <len> <ASCII…> (len 1–31) for one-off names like E, B, rho, alpha, 1.23e-4.
    - (Optional) **macro call**: 0xE0 <macro-id>; **macro def** lives in a header dictionary (rarely needed for single equations).
3. **(Optional) compress**
    - If payload > a few dozen bytes, apply **DEFLATE** (zlib) to the opcode stream.

- Set a **flags** bit so the decoder knows to inflate.

4. **Wrap with a tiny header + checksum** (still inside QR's Byte mode)
5. magic="MQ" (2B) | ver=1 (1B) | flags (1B) |
6. dict_len (1B) | dict entries… | payload_len (2B) | payload… | sha1_7 (7B)
    - sha1_7 is the first 7 bytes of SHA-1 over everything before it—fast integrity check.
    - dict entries only present if you override or add tokens; otherwise dict_len=0.

Typical size wins: Ampère–Maxwell Cr(B)=m0J+m0e0Dt(E)
raw UTF-8 ≈ 19 B → opcodes ≈ 12–16 B → often ≤ 16 B after DEFLATE header amortization (compression may be skipped for tiny strings).

QR pipeline: version/EC selection, block interleave, masking, format/version bits

Once you have the **byte payload**, you use a *standard* QR encoder (Model 2, Byte mode). Nothing custom here:

1. **Choose version & error correction (EC)**
    - Version = matrix size (1–40).
    - EC level = L/M/Q/H (higher EC ⇒ fewer data bytes but more robust).
    - Pick the **smallest version** that fits your *payload bytes* at the chosen EC.
2. **Build the data codewords**
    - Add QR's terminator & pad bytes (0xEC, 0x11) until the capacity is filled.
3. **Reed–Solomon parity**
    - Split data codewords into the version/EC-specified blocks.
    - Compute RS parity per block over GF(256) (standard generator).
    - **Interleave** data codewords across blocks, then interleave parity codewords—this spreads damage.
4. **Place patterns & write bits**
    - Draw **finder**, **separator**, **timing**, **alignment** patterns; reserve **format** and **version** areas.

    - Write codewords in the standard **zig-zag** (two columns at a time, up/down), skipping reserved modules.

5. **Masking**
    - Try all **8 masks**; compute penalty (N1–N4 rules: runs, 2×2 blocks, finder-like patterns, balance).
    - Apply the **lowest-penalty** mask.
6. **Format & version info**
    - Write the 15-bit **format** string (EC + mask, with BCH).
    - For version ≥ 7, write the 18-bit **version** string (with BCH).

Result: a fully standard QR. Any phone camera returns your bytes, which your decoder turns back into the canonical **MathString**.

Capacity planning & byte counts (single vs bundle)

How to estimate quickly

1. **Count bytes** of your MathQR payload *before* QR (header + dict + payload + checksum).
2. Pick a **target EC** (H for stickers/posters; Q/M for tight papers).
3. Ask the QR library for the **smallest fitting version** (it knows the capacity table).

Rules of thumb (practical)

- **Single equations** (Maxwell, heat, Schrödinger with no macros):
    - **EC = H** usually fits in the **smallest versions**.
    - Expect **v1–v2** for very short strings; **v2–v3** if you include metadata.
- **Small bundles** (e.g., all four Maxwell lines + Maxwell4: label):
    - **EC = H/Q** typically **v3–v4**; with compression, rarely larger.
- **Lecture labels** (dozens of short theorems across a deck):
    - Per-theorem QRs at **EC = H**; or a per-slide bundle at **EC = Q**.

Concrete byte sketches

- **Ampère–Maxwell, opcodes only:** ~14 B + header (~12 B) + checksum (7 B) ≈ **33 B**.

- **All four Maxwell + newlines:** raw ≈ 70–90 B; opcodes often compress to **<60 B**; with header/checksum **~80–90 B** → still small QR with EC=H.

Tip: For tiny payloads (< ~25 B), skip compression—DEFLATE's header can outweigh gains. For multi-line bundles, compression is usually worth it.

Minimal encoder pseudocode (end-to-end)

```
def mathqr_bytes(ultra: str, macros=None, dict_overrides=None, compress=True):
    toks = tokenize_ultra(ultra)              # deterministic tokenizer
    stream = bytearray()
    for t in toks:
        if t in OPCODES: stream.append(OPCODES[t])
        elif is_identifier_or_number(t):
            b = t.encode('ascii'); stream += bytes([0xF0, len(b)]) + b
        else:
            raise ValueError("unknown token: "+t)
    payload = zlib.compress(bytes(stream)) if (compress and len(stream)>24) else bytes(stream)
    header  = b"MQ" + bytes([1, (1 if compress else 0)])
    dictsec = build_dict(dict_overrides or {})
    body    = len(payload).to_bytes(2,'big') + payload
    sig     = hashlib.sha1(header+dictsec+body).digest()[:7]
    return header + dictsec + body + sig


# Then pass to any QR library in Byte mode with your chosen EC.
```

Decoder path: scan → bytes → verify magic/version → check sha1_7 → inflate if flagged → decode opcodes → **UltraString** → (expand macros if present) → **MathString**.

That's the whole pipeline: tiny, deterministic payloads riding on a rock-solid, standard QR stack.

Decoding & Round-Trip

Phone/desktop decode → canonical string

Path: *QR scan → bytes → MathString (canonical)*

1. **Scan** with any standards-compliant reader (phone camera, desktop app) in **Byte mode**.
2. **Validate header**: expect magic="MQ", ver=1, flags. Reject if missing.
3. **Read dictionary** (if any): dict_len then dict entries. Entries map opcodes to ASCII tokens; if absent, use the default token table.
4. **Read payload**: payload_len + payload_bytes. If flags.compress=1, **inflate** with DEFLATE.
5. **Verify integrity**: recompute sha1 over (header || dict || body) and compare the first 7 bytes to sha1_7. If mismatch, report **corrupt payload**.
6. **Decode opcodes → UltraString**: apply the fixed token table and dictionary overrides; identifiers/numbers come from length-prefixed atoms.
7. **Normalize to MathString**: expand spacing, precedence parentheses, and long names (L2→Lap, Dt→d/dt, etc.) to produce the single, deterministic **MathString**.

Pseudocode (decoder core)

```
b = scan_qr()                    # bytes from any QR scanner
assert b[:2] == b"MQ"
ver, flags = b[2], b[3]
ptr = 4

dict_len = b[ptr]; ptr += 1
dict_override = read_dict(b, ptr, dict_len); ptr += dict_bytes(dict_len)

paylen = int.from_bytes(b[ptr:ptr+2], 'big'); ptr += 2
payload = b[ptr:ptr+paylen]; ptr += paylen
sig = b[ptr:ptr+7]
```

```
assert sha1(b[:ptr])[:7] == sig


if flags & 0x01: payload = zlib.decompress(payload)
tokens = decode_opcodes(payload, dict_override)   # → token sequence
ultra  = tokens_to_ultrastring(tokens)          # → UltraString
mstr   = normalize_to_mathstring(ultra)         # → MathString (canonical)
return mstr
```

Optional: macro expansion, metadata readout

Macros. If the payload includes a macro table (flagged in flags), the decoder may:

- **Expand** macros to emit a fully explicit MathString (best for archival/search), or
- **Preserve** macros in UltraString for re-encoding with minimal size (best for re-printing QRs).

Either way, the **checksum** is computed against the canonical MathString policy selected by the encoder (expand vs keep), so round-trip integrity remains testable.

Metadata. If present (as a tiny CBOR/JSON block), parse non-destructively:

- id, ns, unit, roles, tags, notes, trace, sha1 (full).
- Display to users, attach to filenames, or store as sidecar .json.
- Never let metadata change the MathString; it's advisory.

Failure modes & recovery

- **Unknown token/opcode**: if a dict override is included, use it; otherwise emit a **stub token** and continue, marking the output as *lossy*.
- **Bad checksum**: warn the user and still show the decoded UltraString (useful if print damage is minor).
- **Compression error**: advise rescanning at higher contrast; fall back to raw payload if the flag is inconsistent.

Integration with renderers (glyphs) or analysis tools

Once you have the canonical **MathString**, you can fan out:

1) Renderers (human-facing)

- **TeX/MathML generator**: deterministic pretty-print for papers.
- **Glyph cards**: factor (subject | operator | object) lanes and render pictograms; attach accuracy/units badges if @[num ...] exists.
- **SEMTILES**: token-tile murals for compact, label-like visuals.

2) Analysis (machine-facing)

- **Metric extractor**: compute operator alphabet size, lane separability, K/E split, locality, linearity, duality, continuity detect (our ORH features).
- **Equivalence checker**: normalize and hash; quick duplicate detection via sha1.
- **Corpus tools**: cluster/UMAP by feature vectors; search by token subsequences or roles (e.g., "find all with Cr and Dt present").

3) Workflow hooks

- **CI/Lint**: verify MathString determinism (re-normalize and compare), check that metadata sha1 matches content.
- **Round-trip QA**: encode → QR → decode → compare sha1 and byte-for-byte MathString. Flag drift immediately.
- **Structured append**: for long payloads spanning multiple QRs, stitch parts by part_index/part_total fields; verify all parts share the same sha1.

UX suggestions

- On desktop, decoding shows **(a)** the MathString, **(b)** optional TeX preview, **(c)** metadata panel, **(d)** a "Copy UltraString" button.
- On mobile, show MathString with a **"Share as text"** action; defer heavy rendering to a server if needed.

Security & privacy

- Payload is **opaque bytes** treated as data, not code.
- Enforce **size caps** (e.g., ≤10 KB per QR) and disable execution of any embedded content.

- Keep macro expansion **pure** (no I/O), and reject non-ASCII identifiers if policy requires.

Bottom line: any phone can capture the math; any machine can restore the **same** canonical statement, verify it via checksum, and immediately reuse it—for display, analysis, or re-print—without human retyping or format drift.

Case Studies

Maxwell (four equations; single-code bundle vs per-equation codes)

UltraStrings

Dv(E)=rho/e0

Dv(B)=0

Cr(E)=-Dt(B)

Cr(B)=m0J+m0e0Dt(E)

Per-equation codes.
Each line fits comfortably in small QR versions at **EC=H**. Typical MathQR payload (header+payload+checksum) is ~30–40 bytes, producing **v1–v2** matrices. Use these when labeling apparatus, slides, or sections—readers can scan exactly the equation they're looking at.

Single-code bundle.
Concatenate with a short label and newlines:

Maxwell4:

Dv(E)=rho/e0

Dv(B)=0

Cr(E)=-Dt(B)

Cr(B)=m0J+m0e0Dt(E)

Bundle payload is usually **≤100 bytes** with minimal benefit from compression; expect **v3–v4** at **EC=H**. This is ideal for a paper figure or a poster where one scan reveals the full set.

Figure 1. Maxwell’s Equations

Figure 2. Maxwell’s Equations

Dt(u)=alphaL2(u)

Payload ~25–35 bytes total → **v1–v2, EC=H**.

Schrödinger (macro wins)
Define a local Hamiltonian macro to cut bytes:

@H(psi)=(-hb^2/(2m))L2(psi)+Vpsi

ihbDt(psi)=H(psi)

- **With macro kept** in UltraString: payload often **≤40–50 bytes** total (v2–v3, EC=H).
- **With macro expanded** (no macro table): ~70–90 bytes (v3–v4, EC=H).

Recommendation
For single equations with a heavy recurring part (e.g., Hamiltonians, constitutive laws), keep the macro in the UltraString payload (flagged so decoders know to expand or preserve it). For archival bundles, consider expansion for full self-containment.

Practical printing guidelines (module size, quiet zone, contrast)

Module size (the little squares)

- Journals/handouts: **0.8–1.2 mm** per module.
- Posters: **1.5–2.5 mm** per module so phones focus quickly.
- Tiny stickers: go **EC=H** and ≥ **1.0 mm** modules.

Quiet zone

- Keep a white border **≥ 4 modules** around the code on all sides. Avoid crop marks or background textures intruding into this area.

Error-correction level (EC)

- **H** (≈30% recovery) for anything that may be photographed at an angle, near a logo, or on matte paper.
- **Q/M** if space is extremely constrained and you control the environment (clean print, flat page).

Contrast & color

- Highest reliability: **black on white**.
- If branding colors are required, ensure a **contrast ratio ≥ 4.5:1** and avoid low-saturation light colors (especially for the modules).
- Do **not** invert (white modules on dark background) unless you've tested with target scanners.

Surface & finish

- Glossy surfaces can glare under overhead lights; **matte** yields better first-try scans.
- Avoid halftone screens behind the code; they can alias with modules.

Placement & size budgeting

- Allow **1.0–1.2×** the code width for caption text beneath.
- If embedding multiple QRs in a grid (e.g., the 2×2 Maxwell set), keep **≥ 1 cm** gutters between codes to prevent "cross-scan."

Pre-press checklist

- Print at **300–600 dpi** (vector preferred).
- Verify a **test scan** with a stock phone camera (no special app).
- If using the bundle QR, verify newlines and label ("Maxwell4:") decode exactly—some scanner UIs wrap long text; your decoder should still receive the raw bytes intact.

What the reader experiences

- **Scan → text:** Any camera app returns the canonical **UltraString** (or bundle).
- **Tap → share/copy:** Readers can paste into your decoder or into the paper's supplemental materials.
- **Round-trip integrity:** Our checksum detects damage; if present, the decoder reports and still shows the best-effort UltraString for recovery.

This section, combined with Figures 2–5 as noted, demonstrates both **per-equation** precision and **bundle** convenience, plus the concrete production settings that make MathQR reliable in the wild.

Related Approaches

Plain-text / TeX in QR (what exists)

QR as dumb transport.

- The most common practice is to drop **raw text** (including TeX snippets) or **URLs** into a standard QR. It works everywhere: any phone returns the same byte string that went in.

- TeX-in-QR is widely used in papers/posters (e.g., \int_0^\infty e^{-x^2}dx=\frac{\sqrt{\pi}}{2}), or to link to a LaTeX source, arXiv page, or a code repository.
- Some workflows add **generic compression** (e.g., DEFLATE) before QR to save space, but the payload is still treated as opaque text.

Limits of plain-text/TeX in QR.

- **No canonicalization:** the *same math* written with different TeX macros or spacing yields different bytes, so hashing, dedup, and robust identity checks are weak.
- **Ambiguity & verbosity:** TeX is layout-oriented; equivalent formulas may serialize very differently and are relatively long (hurts small QR versions, especially at high ECC).
- **No math structure:** a scanner returns a string; any semantic recovery (e.g., identify operators vs identifiers) is a separate, brittle parsing problem.

Other 2D codes.

- **Data Matrix, Aztec, PDF417** offer similar "opaque payload" behavior with different capacity/robustness trade-offs. They don't add math semantics either.

Math on paper without QR.

- **Unicode math** gives compact, human-friendly symbols but is not reliably copy-safe (font/normalization/selection issues) and still lacks structural guarantees.
- **OCR pipelines** (e.g., scanning equations from slides) recover text post hoc, but accuracy and structure are fragile.

Math-aware compaction (what's new here)

Canonical, copy-safe string first.

- We define a **MathString/UltraString** that normalizes spacing, tokens, and precedence so *the same math* always yields the **same bytes**—independent of local TeX macros or typographic choices.

Domain tokenization instead of raw text.

- A tiny, fixed **math opcode dictionary** (Dv, Cr, Dt, L2, Gd, … + punctuation/constants) maps directly to bytes. Identifiers/numbers use short length-prefixed atoms.
- This achieves **size comparable to compressed text**, but with **deterministic semantics** out of the QR—no heuristic parsing.

Optional macros with policy.

- Repeated heavy sub-expressions (e.g., Hamiltonians) can be macro'd with an explicit, scoped table. The encoder records whether macros are **kept** (for density) or **expanded** (for archival). Either way, decoding is unambiguous.

Metadata & integrity baked in.

- A tiny header carries **flags**, optional **dictionary overrides**, and a **checksum** (e.g., 7-byte SHA-1 prefix). Decoders can prove round-trip identity of the math, not just the text.

Still 100% standard QR.

- After pre-encoding, we use normal **Model-2 QR** (Byte mode, RS ECC, masking). Any phone camera works; our decoder adds the math-aware step.

Round-trip guarantees and downstream use.

- Scan → bytes → **canonical MathString** every time. From there we can:
    - render (TeX/MathML, glyph cards),
    - analyze (operator metrics),
    - archive (stable IDs/hashes),
    - or re-encode (same bytes, same QR).

Human-learnable visuals (optional).

- Alongside the machine-readable QR, we provide **SEMTILES/glyphs** so humans can recognize families at a glance—without affecting the QR payload.

How this complements prior work

- If you already print **TeX in QR**, you can drop in **MathQR** to gain **determinism, smaller payloads**, and **immediate structure** on decode.
- If you rely on **links**, MathQR can embed both a **canonical math payload** and a **URL** (or split across multiple QRs with structured append).
- If you use **Unicode math**, MathQR preserves visual economy by emitting a **compact UltraString**, but removes the portability pitfalls.

Bottom line: previous approaches treat math as ordinary text; **MathQR treats math as math**—canonicalized, tokenized, checksummed—while staying fully compatible with the QR ecosystem you already have.

Discussion

Interop with LaTeX/MathML; archives & DOIs; classroom and lab uses

LaTeX/MathML. MathQR decodes to **MathString**, from which we can deterministically emit LaTeX or MathML. That gives a clean bridge: authors draft in MathString; publishers render LaTeX/MathML for layout; round-trips never drift because MathString is the source of truth. In practice: (1) decode QR → MathString, (2) ms2tex/ms2mathml for display, (3) keep MathString alongside the PDF for reproducibility and search.

Archives & DOIs. Include MathString in article metadata (e.g., as a small JSON field), attach a **checksum** and an optional **persistent ID** (DOI or ARK). Two patterns work well:

- **Inline:** a QR in the figure encodes MathString; the figure caption lists the checksum; the supplementary CSV maps checksums → human titles.
- **Linked:** QR payload bundles {doi, ms, sha1} so scanning recovers both the canonical math and the canonical record. Re-hosting never severs the equation's identity.

Classroom & lab. Stick per-equation QRs on whiteboards, handouts, benches, and apparatus. Students scan → canonical text (copy/paste into CAS or notes). In labs, label instruments with governing equations/units as **permanent stickers** (EC=H, larger modules). For assessments, instructors can randomize parameters but keep a stable MathString "skeleton" to check structure automatically.

Limits (very long proofs), structured append, future token sets

Limits. Very long objects (multi-page derivations, full proofs) exceed practical QR capacity/scan UX. MathQR is best for **equations, identities, short systems, and small macros**. For long content, embed a **pointer** (DOI/URL) plus a **hash of the MathString pack**, keeping integrity without bloating the code.

Structured append. When you must exceed a single code:

- Use QR's **structured append** (index/total + parity). Each tile carries part=i/total and the same sha1. Decoders stitch parts, verify the shared checksum, then decode normally.
- Design: cap at ~16–26 parts for usability; print parts in reading order with clear labels; add a small "all-in-one URL" fallback.

Future token sets. The core dictionary (Dv, Cr, Dt, L2, Gd, …) covers a wide PDE/lin-algebra slice. We can grow via:

- **Domains:** tensors (∇·σ, Christoffels), probability (KL, H, logit, softmax), optimization (KKT variants, cones), category theory (Hom, Nat, Lim, Colim).
- **Versioning:** bump the **MathQR header version** and pack new tokens in the per-payload dictionary so older decoders still work (they'll see identifiers instead of unknown opcodes).
- **Localization without drift:** keep tokens ASCII and language-agnostic; provide UI labels per locale while preserving the same bytes.

Operational guidance. Favor **EC=H** for field use; test at device distances (arm's length for A4; 1–2 m for posters). For bundles, compress; for single short lines, skip compression (DEFLATE headers can dominate). Always verify **sha1** after round-trip (encode → print → scan → decode).

Outlook. MathQR doesn't replace LaTeX; it complements it with a **portable, canonical identity** for equations—easy to print, scan, diff, cite, and analyze. As token sets broaden and structured-append UX improves, QR-labelled math can become a low-friction backbone for pedagogy, lab practice, and archival integrity.

Appendix

A. Token dictionary (core v1)

Delimiters & punctuation

- ( ) [ ] { } , : ;

Arithmetic & calculus operators

- Infix: = + - * / ^
- Differential operators (long names; UltraString abbreviations in **bold**):
    - grad (**Gd**), div (**Dv**), curl (**Cr**), Lap (**L2**)
    - d/dt (**Dt**), d/dx (**Dx**), d/dy, d/dz
- Linear algebra:
    - diag, trace, transpose ' (postfix), pseudoinverse ^+

Logic & relations

- Logical: and or not -> <->
- Relations: != < <= > >= in notin subset subseteq

Probability & statistics (optional set)

- P( ) E[ ] Var[ ] Cov[,] ~

Constants & physical symbols (short ASCII)

- pi e i hb e0 m0 c G Lam

Binders & headers

- forall exists exists!
- Section labels: context: claim: hint: ref:
  (UltraString: C: for context, K: for claim; optional)

Identifiers & numbers

- Identifier: [A-Za-z][A-Za-z0-9_]* (ASCII only in v1)
- Number: integer 0|[1-9][0-9]*, decimal d.d+, scientific d(.d+)?[eE][+-]?d+, rational p/q

UltraString abbreviations (lossless)

Gd=grad Dv=div Cr=curl L2=Lap Dt=d/dt Dx=d/dx

hb=hbar e0=eps0 m0=mu0   Lam=Lambda

&=and |=or

B. Minimal grammar (canonical MathString)

Lexical

- Whitespace collapses to a single space; no spaces inside delimiters; exactly one space around top-level infix ops.

EBNF (core expression)

```
Expr      := LogicExpr


LogicExpr := RelExpr ( ( "and" | "or" | "->" | "<->" ) RelExpr )*

RelExpr   := AddExpr ( ( "=" | "!=" | "<" | "<=" | ">" | ">=" | "in"

                  | "notin" | "subset" | "subseteq" ) AddExpr )*

AddExpr   := MulExpr ( ( "+" | "-" ) MulExpr )*

MulExpr   := PowExpr ( ( "*" | "/" ) PowExpr )*

PowExpr   := Postfix ( "^" PowExpr )?   // right-associative

Postfix   := Primary ( "'" | "^+" )*    // transpose, pseudoinverse

Primary   := atom

        | ident "(" ArgList? ")"

        | OpCall

        | "(" Expr ")"

        | "[" Expr ("," Expr)* "]"

        | "{" SetBody "}"
```

```
ArgList   := Expr ( "," Expr )*

OpCall    := "grad" "(" Expr ")"
          | "div"  "(" Expr ")"
          | "curl" "(" Expr ")"
          | "Lap"  "(" Expr ")"
          | "d/dt" Expr
          | "d/dx" Expr | "d/dy" Expr | "d/dz" Expr


SetBody   := Expr "|" LogicExpr

atom      := number | ident | const

ident     := [A-Za-z][A-Za-z0-9_]*

const     := "pi"|"e"|"i"|"hb"|"e0"|"m0"|"c"|"G"|"Lam"

number    := INT | DEC | SCI | RATIO
```

Headers (optional, canonical order)

```
Stmt    := Header* "claim:" Expr ( "and" Expr )*

Header  := "context:" CItems | "hint:" Text | "ref:" Text

CItems  := CItem ( "," CItem )*

CItem   := Binder | Decl

Binder  := "forall" ident ("in" Expr)?
        | "exists!" ident ("in" Expr)?
        | "exists" ident ("in" Expr)?

Decl    := ident "in" Expr
```

UltraString (minified)

- Same AST; remove spaces; abbreviate operators; allow ; as clause separator; multiplication by juxtaposition; headers optional (C:, K:).

Precedence (from tightest to loosest)

1. Function call/indexing
2. Power ^
3. Unary minus / not
4. * / (and juxtaposition in UltraString)
5. + -
6. Relations
7. Logic

Canonicalizer inserts *minimal parentheses* to preserve meaning.

C. Reference encoder pseudocode (MathQR v1)

Overview

- Input: UltraString (or MathString normalized then minified)
- Output: Byte payload ready for any standard QR (Model 2, Byte mode)

1) Tokenize deterministically

```
def tokenize(ultra: str) -> list[str]:
  # greedy match of multi-char ops (->, <->, <=, >=), then single chars,
  # else identifier/number chunk. No spaces exist in UltraString.
  ...
```

2) Map tokens to opcodes

Reserve single-byte opcodes for frequent tokens.

Example opcode table (excerpt)

```
0x00: reserved
0x10: "Gd"   0x11: "Dv"   0x12: "Cr"  0x13: "L2"  0x14: "Dt"
0x18: "Dx"   0x19: "hb"   0x1A: "e0"  0x1B: "m0"  0x1C: "Lam"
0x20: "="    0x21: "+"    0x22: "-"   0x23: "*"   0x24: "/"
```

0x28: "(" 0x29: ")" 0x2A: "[" 0x2B: "]" 0x2C: "{"

0x2D: "}" 0x2E: "," 0x2F: ":" 0x30: ";" 0x31: "<"

0x32: "<=" 0x33: ">" 0x34: ">=" 0x35: "!=" 0x36: "->"

0x37: "<->" 0x38: "&" 0x39: "|" 0x3A: "^" 0x3B: "'"

0xF0: ATOM # length-prefixed ASCII identifier/number (1..31)

0xE0: MCALL # macro call: 1 byte macro id

0xE1: MDEF # macro def: id, argcount, len, bytes...

Atom packing

```
def emit_atom(buf: bytearray, s: str):
    b = s.encode('ascii')
    assert 1 <= len(b) <= 31
    buf += bytes([0xF0, len(b)]) + b
```

3) Optional compression (DEFLATE)

```
def maybe_compress(stream: bytes) -> (bytes, bool):
    if len(stream) <= 24:       # tiny ⇒ skip header overhead
        return stream, False
    comp = zlib.compress(stream)
    return (comp if len(comp) < len(stream) else stream,
            len(comp) < len(stream))
```

4) Header + checksum

```
HEADER:
  magic  : 2B = "MQ"
  ver    : 1B = 0x01
  flags  : 1B = bit0:compressed, bit1:has_dict, bit2:has_macros
  dict   : if has_dict: 1B n, then n entries of (1B opcode, 1B len, bytes)
  body   : 2B payload_len (big endian), payload (stream or compressed)
```

sha1_7 : 7B = first 7 bytes of SHA-1 over (magic..end of body)

Assembly

```
def mathqr_bytes(ultra: str, dict_over=None, macros=None) -> bytes:
    toks = tokenize(ultra)
    stream = bytearray()
    for t in toks:
        if t in OPCODES: stream.append(OPCODES[t])
        else: emit_atom(stream, t)
    payload, compressed = maybe_compress(bytes(stream))

    header = b"MQ" + bytes([1, (1 if compressed else 0) |
                            (2 if dict_over else 0) |
                            (4 if macros else 0)])
    dictsec = build_dict(dict_over or {})   # may be empty
    body = len(payload).to_bytes(2,'big') + payload
    sig  = hashlib.sha1(header + dictsec + body).digest()[:7]
    return header + dictsec + body + sig
```

5) Feed to standard QR

- Use any QR library: **Byte mode**, choose **EC level**, let it pick **smallest version** that fits.

D. Reference decoder pseudocode

```
def decode_mathqr(b: bytes) -> dict:
    assert b[:2] == b"MQ"
    ver, flags = b[2], b[3]; ptr = 4

    # dictionary (optional)
```

```
dict_over = {}
if flags & 0x02:
   n = b[ptr]; ptr += 1
   for _ in range(n):
      op = b[ptr]; ln = b[ptr+1]; ptr += 2
      tok = b[ptr:ptr+ln].decode('ascii'); ptr += ln
      dict_over[op] = tok

paylen = int.from_bytes(b[ptr:ptr+2], 'big'); ptr += 2
payload = b[ptr:ptr+paylen]; ptr += paylen
sig = b[ptr:ptr+7]
assert hashlib.sha1(b[:ptr]).digest()[:7] == sig, "checksum mismatch"

if flags & 0x01:
   payload = zlib.decompress(payload)

tokens = []
i = 0
while i < len(payload):
   op = payload[i]; i += 1
   if op == 0xF0:
      ln = payload[i]; i += 1
      tokens.append(payload[i:i+ln].decode('ascii')); i += ln
   else:
      tok = REVERSE_OPCODES.get(op) or dict_over.get(op)
      if tok is None: tokens.append(f"<UNK:{op:02X}>")
```

```
        else: tokens.append(tok)

    ultra = ''.join(tokens)
    mstring = normalize_to_mathstring(ultra)  # spacing, long names, parentheses
    return {"ultra": ultra, "mathstring": mstring}
```

E. Structured append (long payloads)

When content exceeds a single QR:

Payload split

common_header = ... (magic, ver, flags, dict)

for i in range(N):

part_body = 2B part_len + part_bytes_i

part_sig = sha1(common_header + part_body)[:7]

qr_i = common_header + part_body + part_sig

Also include a small **SA trailer** inside part_bytes_i:

sa: { total: N (1B), index: i (1B), master_sha1_7: 7B }

Decoders:

- Collect parts by master_sha1_7, ensure all index seen, glue part_bytes_i in order, then decode once.

F. Test vectors (sanity)

1. **Ampère–Maxwell (UltraString)**

Cr(B)=m0J+m0e0Dt(E)

- Expected token stream: ["Cr","(","B",")","=","m0","J","+","m0","e0","Dt","(","E",")"]
- Typical opcode stream (example):
  13 28 F0 01 42 29 20 1B F0 01 4A 21 1B 1A 14 28 F0 01 45 29
  (F0 <len><ASCII> used for identifiers B J E)

2. **Heat**

Dt(u)=alphaL2(u)

- Tokens: ["Dt","(","u",")","=","alpha","L2","(","u",")"]

3. **Bundle (Maxwell4)**

Maxwell4: Dv(E)=rho/e0\nDv(B)=0\nCr(E)=-Dt(B)\nCr(B)=m0J+m0e0Dt(E)

- Expect compression **true**; verify checksum matches after round-trip.

G. Minimal normalization rules (reference)

- Numbers → canonical: strip leading zeros; scientific lowercase e with sign; rationals reduced.
- Identifiers → ASCII; reject non-ASCII or map via agreed transliteration.
- Operator spacing → exactly one space at top level; none inside function calls.
- Precedence-aware parentheses added only when needed to preserve parsing.

This appendix is sufficient to implement a small, interoperable **MathQR** toolchain: you can tokenize and canonicalize math, emit compact opcodes, wrap them in a verifiable header, and carry them over **standard QR codes**—then deterministically recover **the same** MathString on any device.

References

1. **ISO/IEC 18004**. *QR Code bar code symbology specification.* International Organization for Standardization, 2015.
2. Reed, I. S., & Solomon, G. **"Polynomial Codes over Certain Finite Fields."** *Journal of the Society for Industrial and Applied Mathematics* 8(2), 1960.
3. Bose, R. C., & Ray-Chaudhuri, D. K. **"On a Class of Error Correcting Binary Group Codes."** *Information and Control* 3(1), 1960.
   Hocquenghem, A. **"Codes Correcteurs d'Erreurs."** *Chiffres* 2, 1959.
4. Deutsch, P. **RFC 1951: DEFLATE Compressed Data Format Specification.** Internet Engineering Task Force, 1996.
5. **FIPS PUB 180-4. Secure Hash Standard (SHS).** National Institute of Standards and Technology, 2015. (SHA-1, SHA-2 definitions.)
6. Lamport, L. **LaTeX: A Document Preparation System.** 2nd ed., Addison-Wesley, 1994.
7. *The Unicode Standard.* The Unicode Consortium.
   — UAX #15: **Unicode Normalization Forms.** (Canonical/compatibility normalization.)
8. W3C. **MathML 3.0 Specification.** World Wide Web Consortium Recommendation, 2014.
9. Mac Lane, S. **Categories for the Working Mathematician.** 2nd ed., Springer, 1998.
10. The Univalent Foundations Program. **Homotopy Type Theory: Univalent Foundations of Mathematics.** Institute for Advanced Study, 2013.
11. **ZXing ("Zebra Crossing")**: Multi-format 1D/2D barcode image processing library. (Open-source barcode/QR encoder–decoder.)
12. **libqrencode**: QR Code encoding library. (Open-source C library for standard QR generation.)
13. **qrcode (CTAN package)**: *Embed QR codes in LaTeX documents.* Comprehensive TeX Archive Network entry.
14. **ISO/IEC 16022**. *Data Matrix bar code symbology specification.* International Organization for Standardization.
    ISO/IEC 24778. *Aztec Code bar code symbology specification.*
    ISO/IEC 15438. *PDF417 bar code symbology specification.*
15. Knuth, D. E. **The TeXbook.** Addison-Wesley, 1984. (Foundational typesetting reference.)

Notes: (a) Items 11–13 cite widely used open-source tooling and TeX integration for QR, relevant to implementation and practice. (b) Items 1–5 cover QR structure, BCH/RS codes, compression, and hashing used in MathQR. (c) Items 6–10 ground interoperability with LaTeX/MathML and the CT/HoTT background touched in the paper.