Cent-Exact by Construction: An Empirical Study of an Integer Ledger for Multi-Payer Expense Splitting

Yousof Selim
Sunway University (Lancaster University dual degree), Malaysia

0 minor units created or lost in 3,999,600 synthetic expenses

1. The integer ledger created or lost 0 minor units across 3,999,600 synthetic expenses; rounding each double share left 49.5% of splits off their bill.
2. The fixed tie-break is exact but not neutral: after 10,000 expenses the largest per-person gap averaged 1,348 minor units, against 39 with rotated ties.
3. Paying the simplified plan left 90.0% of groups unsettled with plain payments and 38.6% with the routed bridge; a direct-plan fallback settled all 24,000.
4. Stale or tampered routing data moved a third party's balance in 553 of 18,841 cases; a stricter bench-only check reduced this to zero.

Limits. Synthetic data only, one heavily loaded machine, sampled tests rather than proofs, a non-optimal simplifier, private package source and no independent review.

Abstract

The problem Splitting shared bills looks simple, but rounding, several payers and 'fewest payments' settle-ups can make balances drift, disagree between phones or never reach zero. Small money errors cost a money app its users' trust.

What I did I built the ledger core of my app Bupples on whole integer cents with one splitting rule. I then compared it with a typical floating-point version on about four million seeded synthetic expenses,

	measured its settle-up planner against the true minimum, and stress-tested how it records payments.
What I found	**The integer ledger never gained or lost a cent, while rounding each floating-point share left half of all splits off their bill. Three weaknesses showed up: the fixed tie-break slowly favours some members over others, paying the simplified plan left 39% of groups unsettled unless a fallback guard was added, and stale or altered payment routes could move a bystander's balance.**
Why it matters	It replaces 'the maths is exact' with measured evidence, names three fixes a real money app can ship, and publishes the benchmark code and raw results. The app's source stays private.

Contributions

- A description of an integer ledger design for multi-payer, multi-currency expense splitting, as shipped in a production app.
- A six-experiment evaluation against a naive IEEE-754 implementation of the same logic and an exact rational reference, with 95% intervals and public benchmark code and raw results.
- An exact optimality study of the greedy fewest-transfers simplifier on 54,000 groups of up to ten people.
- Three weaknesses measured for the first time in this design (tie-break bias, unsettled routed payments and third-party movement under stale or tampered routing data), each with a candidate fix evaluated.

Keywords. money arithmetic · integer minor units · floating point · apportionment · debt simplification · property-based testing · Dart · expense splitting

1 Introduction

Shared-expense apps answer one question over and over: who owes whom, and how much? The arithmetic looks trivial. Add up what each person paid, subtract what each person consumed, and settle the difference. In practice this is where such apps quietly lose trust. A 100.00 bill split three ways cannot be paid in exact thirds. Two phones that add the

same expenses in a different order can disagree by a cent. A settle-up plan that should clear everyone can leave someone owing 0.01 indefinitely.

None of these is a crash. Each is a small, visible inconsistency in the number users check most carefully. The problem becomes harder once a bill can have several payers, uneven shares, more than one currency and a "fewest payments" settle-up mode.

This report studies the ledger core of Bupples, a group expense-splitting app that I designed and released on the App Store and Google Play (version 1.4.6 at the time of writing). The core has been extracted into a dependency-free Dart package, `bupples_ledger` 0.1.0, with 2,062 lines of library code and 2,547 lines of tests. The package is private; the benchmarks and raw results are public.

The design rule is that money is cent-exact by construction. Every amount is an integer count of the currency's minor unit. Every division goes through one deterministic remainder rule. Every balance and plan is derived from one pairwise-debt kernel. The rule is old advice: it is the Money pattern [1], and the hazards of binary floating point for decimal quantities are well documented [2][3]. What this report adds is measurement of what the rule prevents, what it costs, and where it still falls short.

Six experiments (E1–E6) answer four research questions. **RQ1** asks whether the design conserves money and gives order-independent results, and how a naive double implementation compares. **RQ2** asks whether a fixed tie-break, exact for each expense, drifts against particular members over many expenses. **RQ3** asks how good the greedy settle-up plan is, and whether payments made against it can be recorded so that the group actually settles. **RQ4** asks what the operations cost.

The answers are mixed. The integer ledger never created or lost a minor unit, and its outputs did not depend on input order. The design also has three weaknesses that had not been measured before: a tie-break bias that grows with the number of expenses, a routed-payment bridge that leaves many larger groups unsettled, and routing data that, when stale or altered, can move the balance of someone outside the payment.

Contributions.

- A description of an integer ledger design for multi-payer, multi-currency expense splitting, as shipped in a production app: one remainder rule, a per-expense net-position debt kernel, clamped payments, a routed-payment bridge with flow validation, and exact rational receipt co-claims.
- A six-experiment evaluation against a naive IEEE-754 implementation of the same logic and an exact rational reference, with 95% intervals. The benchmark code and every raw result are public.

- An exact optimality study of the greedy fewest-transfers simplifier on 54,000 groups of up to ten people.
- Three weaknesses of the design, each quantified and each paired with a candidate fix that was evaluated but is not yet in the package.

---

2 Background and related work

2.1 Floating point and money

IEEE-754 binary floating point [3] cannot represent most decimal fractions exactly, and Goldberg's survey [2] explains how rounding error arises and accumulates. The familiar consequences for money are that `0.1 + 0.2` is not `0.3` and that `(4.35 * 100).toInt()` is 434. Compensated (Kahan) summation [4] reduces accumulated error in long sums.

Compensation does not solve the allocation problem studied here. Even exact rational shares must eventually be rounded to whole minor units, and shares rounded independently do not, in general, add up to the total. The standard engineering answer is to store money as an integer number of minor units, paired with a currency, inside a value type. Fowler catalogues this as the Money pattern and warns that rounding to the smallest unit makes it easy to lose pennies [1].

ISO 4217 [5] gives each currency a minor-unit exponent: 0 for the Japanese yen, 2 for most currencies and 3 for the Kuwaiti dinar and a few others. On native Dart platforms an `int` is a 64-bit integer; on the web it is a JavaScript number, exact only up to 2^{53} [6]. That bounds the safe range of amounts in a web build.

2.2 Apportionment

Dividing an integer total in proportion to weights, so that the parts are integers that sum to the total, is the apportionment problem. Balinski and Young [7] set out the classic methods for legislative seats, including Hamilton's largest-remainder method: give each party the floor of its exact quota, then hand the leftover seats to the largest fractional remainders. They also show that no method satisfies every desirable property at once.

A money splitter faces the same problem, with cents as seats, and any rule must break ties. The apportionment literature mainly concerns a single allocation. An expense ledger applies its rule thousands of times to the same people, so a tie-break that is harmless once can accumulate (Section 6.2).

2.3 Settling debts with few transfers

Given net balances that sum to zero, settling up means clearing them with as few transfers as possible. Verhoeff [8] presents several variants of this problem as an invitation to computing science and shows that one of them is NP-hard. Pătcaş and Bartha [9] describe the debts' clearing problem as NP-hard in the strong sense and propose an evolutionary heuristic for large instances.

Section 6.3 uses a standard characterisation. Suppose k people have non-zero balances, and p is the largest number of disjoint groups, each summing to zero, into which they can be partitioned. Then the minimum number of transfers is k − p. Each zero-sum group of size s can be cleared with s − 1 transfers. Conversely, the transfers of any plan form a graph in which every connected component sums to zero, and a component spanning s people needs at least s − 1 edges. The common greedy method repeatedly matches the largest debtor with the largest creditor. It uses at most k − 1 transfers but can miss zero-sum subgroups.

2.4 Property-based testing

QuickCheck [10] introduced the practice of stating program properties as executable predicates and checking them on many generated inputs. Hypothesis [11] brought the approach to Python with integrated shrinking and a database of failing examples. The `bupples_ledger` tests use a lighter form of the same idea: seeded `Random` generators inside ordinary `package:test` tests, with the seed reported on failure but no automatic shrinking. Property tests sample the input space; they do not prove the properties.

2.5 Positioning

The work above supplies the representation advice, the theory of single allocations, the complexity of debt clearing and the testing technique. I am not aware of published measurements of how these pieces behave together in a running ledger. Three questions in particular appear unmeasured: what repeated apportionment does to individual members over time; how far a greedy settle-up plan is from optimal on different debt structures; and whether payments made against such a plan can be recorded without losing money or disturbing third parties. This report is a measurement study of one implementation. It proposes no new algorithm, and its findings describe this design under synthetic workloads.

3 Design

3.1 Integer minor units and parsing

An amount is a Dart `int` counting the smallest unit of its currency: 12.50 is stored as 1250, JPY 1,200 as 1200 and KWD 1.500 as 1500. The exponent comes from a small ISO 4217 table [5] and affects only parsing and formatting. The ledger never needs to know which currency it is adding up, because amounts in different currencies are never added together.

Parsing works on the string and never builds a `double`. It keeps the first e fractional digits and rounds half up on the next one, so `"1.005"` becomes 101. Negative input is refused rather than silently stripped. A locale-aware front end resolves grouping and decimal separators from the shape of the string, and refuses the genuinely ambiguous case (`1.234` in English) with a typed error instead of guessing.

3.2 One remainder rule

All proportional splitting goes through one function, `distributeCents(total, weights)`:

1. sort the ids, so that map insertion order cannot matter;
2. give each id $\lfloor \text{total} \times w_i / W \rfloor$, computed with `BigInt` so that the product cannot overflow;
3. hand out the leftover units, always fewer than the number of ids, one at a time: heaviest weight first, ties broken by ascending id.

A negative total (a refund) is the exact mirror of the positive split. The shares therefore always sum to the total, each share is its floor or one more, and the result is a pure function of the set of (id, weight) pairs.

The rule is deliberately not Hamilton's, because "the biggest share rounds up first" is easy to explain to a user. Receipt allocation, where the weights are exact rational claims, does use largest remainder (Section 3.6). Equal splits use the same rule with unit weights, so the leftover units always go to the ids that sort first. Section 6.2 measures what that costs.

3.3 The ledger model

An expense has a total, a payer map (one or more people who paid, summing to the total), a split method (equal, exact amounts, basis-point percentages or share counts) and a currency code. For each expense the kernel computes every participant's net position,

which is the amount paid minus the amount consumed. Creditors and debtors within that expense are matched greedily in id order, and the resulting edges are added to a pairwise map `owes[debtor][creditor]`.

Working from per-expense net positions matters for multi-payer bills. Spreading each consumer's gross share across all payers would double-count money that a payer spent on their own share. Someone who paid exactly their own share nets to zero and appears in no transfer.

Accepted payments are subtracted from the matching pair and clamped at zero, so a duplicate acceptance, or a bill edited down after it was paid, cannot create a debt in the opposite direction. Pending payments move nothing in the confirmed ledger; they are applied, again clamped, only in the payer's own projection. Net balances, the direct plan and the simplified plan are all derived from this one kernel, so they cannot disagree. A multi-currency group is partitioned by currency, and each partition is an independent ledger; nothing is converted.

3.4 Two settle-up plans

The direct plan nets each pair into at most one transfer. Nobody is asked to pay a person they do not owe, so every line is easy to explain. The simplified plan runs the greedy method of Section 2.3 (largest creditor against largest debtor, ties by id) on the net balances. It needs at most k − 1 transfers, but it may route money between two people who share no debt.

3.5 Routed payments

Suppose c owes b 50 and b owes a 50. The simplified plan says "c pays a 50". If that payment is recorded against the pair (c, a), the clamp finds no debt there and the payment vanishes: the money has moved in real life but not in the ledger.

The bridge, `decomposeRoutedPayment`, runs when the payment is recorded. It decomposes the payment into a flow through the current debt graph, taking the direct edge first and then repeated deterministic depth-first paths (largest debt first, ties by id), and stores the resulting covered edges on the payment.

When the ledger applies the payment, it first checks the edge list with `validCoveredEdgeFlow`. The list must contain between 1 and 24 positive edges; every intermediate person must have equal inflow and outflow; the payer's net outflow must equal the receiver's net inflow; and the total must not exceed the payment. If the check passes, each covered edge is reduced, with clamping. Otherwise the ledger falls back to

the plain clamp. In the intended case, flow conservation means that only the payer's and the receiver's balances change.

3.6 Receipt co-claims

Receipt splitting lets each person claim items, including rational shares of a unit such as half a pizza or a third of a jug, with numerators and denominators up to 99. Claims are clamped to each item's quantity in a fixed order. Each item's value is divided among its claimants with exact `BigInt` fractions and largest remainder. Tax and service are spread over every item, claimed or not, so a small claim on a large receipt carries only its own share. The package maintains two identities for every input: `Σ subtotals + unclaimedPot = Σ items` and `Σ tax-and-service shares + unallocated = tax + service`.

---

4 Implementation

`bupples_ledger` is pure Dart (SDK ^3.11) with no runtime dependencies and no I/O, in eleven library source files. The largest are `receipt_claims.dart` (600 lines), `ledger.dart` (353) and `routed_payment.dart` (202). It was extracted from the Flutter app's domain code in `lib/features/settlement/domain/` (`ledger.dart`, `cents.dart`, `simplified_decomposition.dart`) and `lib/features/receipts/domain/receipt_allocation.dart`. The app's git history dates the multi-payer net-position ledger to 24 June 2026 and the routed-payment fix to 28 July 2026.

In the app, pair balances and most settle-up screens use the direct plan. Where a routed payment is offered, the settle-up sheet sizes the payment by the flow that actually decomposes into real debts, and falls back to the direct edge when none does. I did not evaluate these app flows; everything below concerns the package.

The test suite has nine files and 153 tests, all of which passed under `dart test` on the Dart VM on 1 November 2026. Unit tests carried over from the app pin worked examples, including a known limitation: a simplified row across disconnected debts cannot be applied. A property file checks the invariants on seeded generators, with 100 to 500 seeds per property:

- shares sum to the total, and each share is its floor or one more;

- insertion order does not matter, and refunds mirror charges;
- zero-, two- and three-decimal currencies round-trip through parsing and formatting;
- balances sum to zero, and both plans clear them;
- routed payments never create money or flip the sign of a balance.

The experiments live in a separate package, `cent_exact_bench`: about 1,700 lines of Dart and a 210-line Python plotting script. It adds three things that are not part of the library: a naive double reference implementation, two alternative remainder rules, and two candidate mitigations for routed payments. It was run against a snapshot of the library whose SHA-256 sums are published with the artefact.

---

5 Study design

5.1 Research questions

- **RQ1 (exactness and determinism).** Does the integer design conserve money and give the same outputs regardless of input order and platform? How often does a naive double implementation of the same logic fail, and how far does it drift?
- **RQ2 (fairness of the remainder rule).** Does a fixed tie-break, exact for every expense, drift against particular members over many expenses, and does rotating the tie-break help?
- **RQ3 (settle-up plans).** How far is the greedy plan from the minimum number of transfers? Can payments made against it be recorded so that the group actually settles, and is the routing data robust to staleness and tampering?
- **RQ4 (cost).** What do the main operations cost at realistic group sizes?

Table 1. Experiments and the research questions they answer.

Experiment	What it measures	Inputs	RQ
E1	Conservation, drift and tie-break bias against doubles and an exact rational reference	1,440 ledgers, 3,999,600 expenses	RQ1, RQ2
E2	Greedy and direct plans against the exact minimum number of transfers	54,000 ledgers	RQ3
E3	Paying the simplified plan under five recording strategies; tamper and staleness campaign	24,000 ledgers; 18,841 tamper cases	RQ3
E4	Order independence under shuffling; Dart VM against JavaScript output	40,000 shuffled runs; 500-case digest	RQ1
E5	CPU time per operation	One ledger per size	RQ4
E6	Exhaustive amount parsing; receipt co-claim conservation	21,000,000 strings; 20,000 receipts	RQ1

5.2 Workloads

All inputs are synthetic. Each comes from `dart:math` `Random(seed)` with a recorded seed formula, so every run is repeatable bit for bit on the same Dart SDK.

The E1 generator draws expense totals log-uniformly between about 1 and 2,000 major units (100 to 200,000 for zero-decimal currencies). Each expense is split among a random non-empty subset of the group: 60% equally, 20% by share counts of 1 to 4, and 20% by percentage (random basis points summing to 10,000). In 30% of expenses two or three people pay uneven parts of the bill. E1 crosses group size $n \in \{2, 5, 10, 20\}$, ledger length $m \in \{10, 100, 1{,}000, 10{,}000\}$ and currency exponent $e \in \{0, 2, 3\}$, with 30 seeds per cell: 1,440 ledgers and 3,999,600 expenses.

E2 and E3 use three ledger generators with 3 to 30 expenses each. Uneven reuses the E1 generator with two decimals. Round uses amounts that are multiples of $10 \cdot n$ major units (up to $100 \cdot n$), split equally among 2 to n people with one payer, so zero-sum subgroups are common. IOUs are one-to-one loans of 10 to 100 major units in steps of 10. E2 uses all three generators with n = 2 to 10; E3 uses uneven bills and IOUs with $n \in \{3, 4, 5, 6, 8, 10\}$. Each cell has 2,000 ledgers.

5.3 Baselines and references

The naive implementation holds amounts as doubles in major units (12.34) and computes each share as `amount × w / W`, with the same participants, weights and payer amounts as the integer run. It has three variants:

- raw: unrounded doubles throughout;
- Kahan: raw, but each person's balance is accumulated with compensated summation;
- round-each: each share rounded to the nearest minor unit independently, as much hand-written split code does.

As ground truth, the bench also computes every balance as an exact rational number, with `BigInt` numerators over the least common multiple of the ledger's weight totals. For E2, the exact minimum number of transfers is k − p (Section 2.3), computed by dynamic programming over subsets of the k non-zero balances in $O(2^k \cdot k)$ time.

5.4 Statistics

Means are reported with Student-t 95% intervals over ledgers; a per-ledger proportion, such as the share of splits that miss their bill, is treated as a per-ledger mean. Proportions of independent trials (ledgers in E2 and E3, shuffles in E4, receipts in E6) are reported with Wilson 95% intervals. When no failure is observed, the Wilson upper bound is given, because a zero count from sampling bounds the failure rate rather than excluding failures. E6's parsing sweep is exhaustive over its domain, so it needs no interval. Pooled figures average over the synthetic mix of group sizes, lengths and exponents; they describe that mix, not any population of real groups.

5.5 Environment

All runs used one Apple M1 Pro with 16 GB of RAM, macOS 27.0.1 and Dart SDK 3.12.1 (stable, macos_arm64). Every experiment except the platform digest was AOT-compiled with `dart compile exe`. The runs took place on 1–2 November 2026. The machine was shared with other heavy jobs throughout (load average about 150–180). The load does not change the outputs of E1–E4 and E6, which are deterministic, but it does affect E5 (Section 6.4).

---

6 Results

6.1 RQ1: exactness and determinism

Conservation. The integer package had no ledger whose balances failed to sum to zero and no expense whose shares failed to sum to its total: 0 of 1,440 ledgers (Wilson upper bound 0.27%) and 0 of 3,999,600 expenses (upper bound 0.0001%). For shares this is guaranteed by the construction of `distributeCents`, so the experiment checks the implementation rather than testing a hypothesis. The informative part of Table 2 is the comparison with the naive variants, which fail in two quite different ways.

Table 2. E1: the integer package against a naive double implementation of the same logic, by ledger length. Means over 360 ledgers per column (30 seeds × 4 group sizes × 3 exponents), with Student-t 95% intervals over ledgers in brackets.

Metric	m = 10	m = 100	m = 1,000	m = 10,000
Integer: ledgers whose balances do not sum to 0	0%	0%	0%	0%
Integer: expenses whose shares ≠ bill	0%	0%	0%	0%
Round-each: expenses whose shares ≠ bill (mean per ledger)	48.5% [46.0, 51.1]	49.7% [47.4, 51.9]	49.5% [47.2, 51.7]	49.5% [47.2, 51.7]
Round-each: ledgers whose displayed balances do not sum to 0	88.6% [85.3, 91.9]	99.7% [99.2, 100]	100%	100%
Round-each: minor units created or lost per ledger	3.8 [3.4, 4.2]	18.8 [17.5, 20.0]	159 [154, 165]	1,559 [1,534, 1,583]
Raw: Σ balances ≠ 0.0 exactly	78.9% [74.7, 83.1]	88.9% [85.6, 92.1]	94.7% [92.4, 97.0]	98.1% [96.6, 99.5]
Raw: mean |Σ balances| (minor units)	5.2×10^{-11}	4.7×10^{-10}	4.6×10^{-9}	4.8×10^{-8}
Raw, each balance rounded for display: Σ ≠ 0	40.3%	40.0%	44.7%	38.6%
Exact rationals, each balance rounded for display: Σ ≠ 0	39.7%	40.0%	44.7%	38.6%

Allocation failure dominates. Rounding each share on its own left 49.5% of all 3,999,600 splits not summing to their bill (pooled; the mean per ledger is 49.3% [48.1, 50.4]), and 99.2% of ledgers contained at least one such split. The mismatch rate rises with group

size: 17.6% of splits at n = 2, 44.1% at 5, 61.3% at 10 and 74.1% at 20. Displayed balances failed to sum to zero in 97.1% of round-each ledgers.

These errors largely fail to cancel, so the money created or lost grows roughly in proportion to ledger length, reaching a mean of 1,559 minor units per 10,000-expense ledger (15.59 major units in a two-decimal currency; Figure 1).

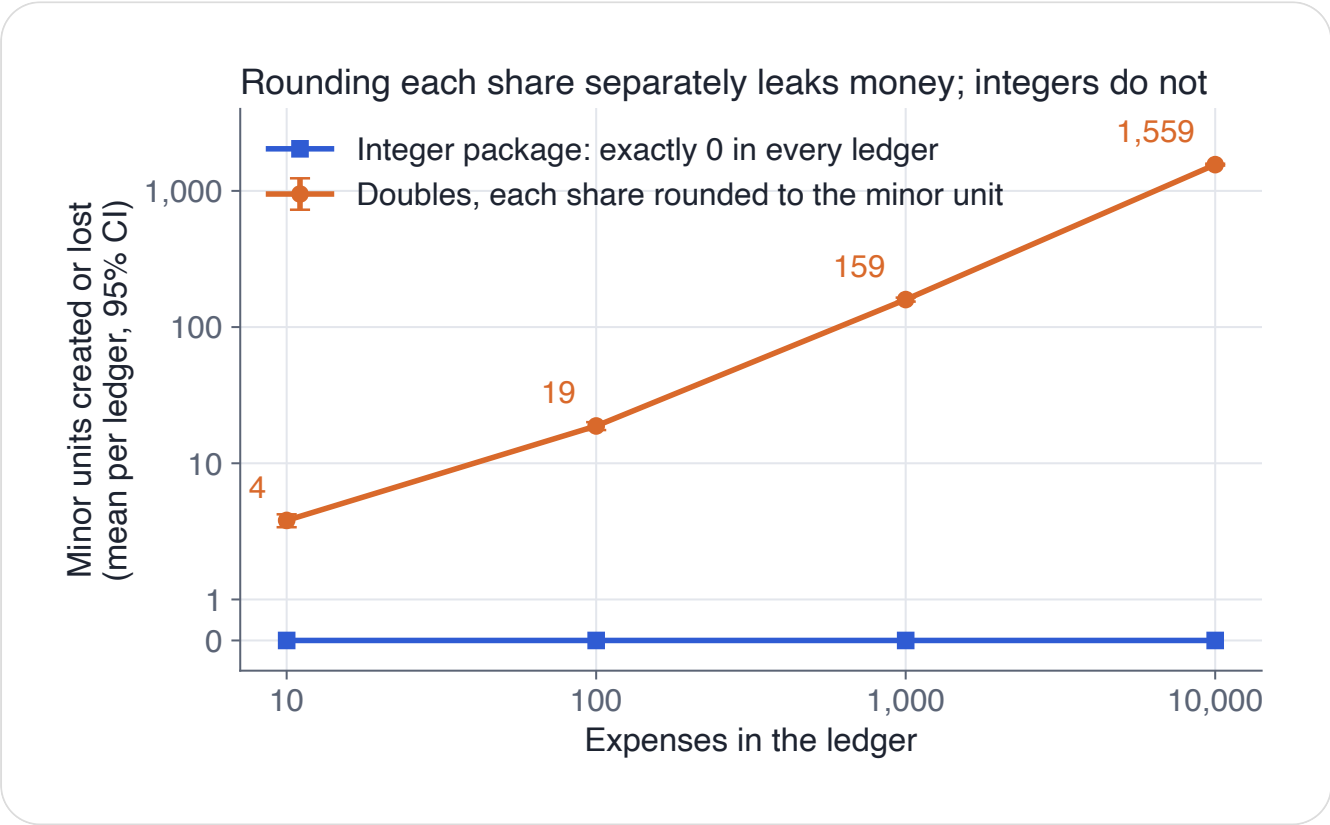


Figure 1. Minor units created or lost per ledger when each share of a double split is rounded separately, against the integer package (exactly 0 in all 1,440 ledgers, drawn on a separate baseline because 0 has no position on a log axis). Error bars are 95% intervals over 360 ledgers per point.

Accumulated floating-point error is real but tiny. Unrounded doubles rarely produced balances summing to exactly 0.0: 90.1% [88.6, 91.7] of ledgers failed an exact equality check. The residual, however, never exceeded 4.5×10^{-7} minor units. The largest per-person error against the exact rational balance, across all 1,440 ledgers, was 3.0×10^{-7} minor units, and Kahan summation reduced that maximum to 1.1×10^{-8}.

The last two rows of Table 2 are the more instructive comparison. When raw balances are rounded for display, they fail to sum to zero in 40.9% of ledgers. Exact rational balances rounded the same way fail in 40.8%. The two rates differ by less than one percentage point at every group size. The largest difference is for two-person groups, where the

exact control never fails and raw doubles failed in 0.8% of ledgers, because the tiny error tipped a balance across a rounding boundary.

The visible problem, then, is not that floating-point values are slightly wrong. It is that any representation which keeps fractional cents must round late and independently, and independent rounding does not conserve. The integer design avoids this by allocating whole minor units when each expense is split, so nothing is left to round afterwards. The results were similar for exponents 0, 2 and 3; per-exponent figures are in the artefact.

Parsing (E6). E6 parsed every two-decimal string from 0.00 to 99,999.99 (10,000,000 inputs). The idiom `(double.parse(s) * 100).toInt()` was wrong for 587,200 of them (5.87%); for example, "0.29" gives 28. Using `.round()` instead was right for all of them, as was the package parser. For three-decimal strings up to 9,999.999 (10,000,000 inputs), `toInt()` was wrong for 94,381 (0.94%). For 1,000,000 strings with one extra digit ending in 5, such as "1.005", `(double.parse(s) * 100).round()` missed the half-up result in 65,628 cases (6.56%); the string-based parser missed none. Because the sweep is exhaustive, these are exact counts over each domain, not estimates.

Receipts (E6). On 20,000 seeded receipts (91,090 item lines, 2–8 people, rational claims from 1/6 to 1), the package's two conservation identities held in every case (upper bound 0.019%). A naive version that values each claim in doubles and rounds each member's subtotal and the unclaimed remainder failed to add up to the receipt in 7,477 receipts (37.4% [36.7, 38.1]).

Order independence and platforms (E4). E4 built 5,000 ledgers with accepted, partial, pending and routed payments, and shuffled each one eight ways: the order of expenses and payments, and the insertion order of every weight map and payer map. For every shuffle, the bench compared net balances, both plans, the viewer projections, the pairwise debt map, the decomposition of every simplified row and every expense's shares. Across 40,000 shuffled runs there were 0 mismatches (upper bound 0.0096%).

The naive double balances differed bitwise from the unshuffled run in 90.8% of shuffles, because floating-point addition is not associative. A displayed cent changed in 237 shuffles (0.59% [0.52, 0.67]), affecting 64 of the 5,000 ledgers; two phones that received the same expenses in a different order would show different amounts in those cases. Finally, a digest of the canonical outputs for 500 cases was identical on the Dart VM and in the `dart compile js` build under Node.js 26 (polynomial hash 132897213 over 1,375,426 characters). This supports, for these inputs only, that the VM and web builds agree.

6.2 RQ2: exact, but not neutral

Exactness per expense says nothing about who absorbs the leftover units over time. For each ledger, E1 also records the largest gap between any person's integer balance and their exact rational balance (Figure 2).

Under the package rule the gap grows linearly with ledger length. Its mean is 2.1 minor units at m = 10, 15.2 [14.7, 15.7] at 100, 137 [134, 140] at 1,000 and 1,348 [1,316, 1,380] at 10,000, about 0.13 minor units per expense. Hamilton's largest remainder with ties broken by id behaves almost identically (1,345 [1,315, 1,375] at m = 10,000), because every remainder ties in an equal split.

A bench-only variant keeps largest remainder but breaks ties by a hash of (expense id, member id). It is still deterministic and order-independent, but the tie order changes from one expense to the next. It reduces the gap at m = 10,000 to 39.3 [37.2, 41.4], a 34-fold reduction, and the gap then grows roughly with the square root of ledger length.



Figure 2. Mean largest per-person gap between the integer balance and the exact rational balance, by ledger length. The package rule and largest remainder with id ties overlap; rotating ties per expense (bench only) grows far more slowly. Error bars are 95% intervals over 360 ledgers per point.

Signed gaps show who pays. In one 20-person, 10,000-expense ledger (`Random(1)`, two decimals), the gaps form a monotone gradient by id. The member who sorts first was charged 1,609 minor units (16.09 major units) more than their exact share, and the member who sorts last 1,572 less. With rotated ties every member stayed within ±50.

In the app, member ids are opaque account identifiers, so which member is disadvantaged is arbitrary, but within a group it is always the same one. For a group with 100 expenses, the largest gap averages about 15 minor units; with 1,000 expenses, about 137. The effect is small, but it is systematic rather than random.

6.3 RQ3: settle-up plans

Simplifier quality (E2). Table 3 and Figure 3 compare the greedy and direct plans with the exact minimum.

Table 3. E2: greedy simplifier against the exact minimum number of transfers, groups of 2–8 people (14,000 ledgers per generator). The +1 and +2 columns count ledgers; brackets are Wilson 95% intervals.

Generator	Greedy optimal	+1 transfer	+2	Mean minimum	Mean greedy	Mean direct	Direct volume ÷ minimum
Uneven	99.94% [99.88, 99.97]	9	0	3.99	3.99	10.75	1.36×
Round	99.21% [99.05, 99.34]	109	2	3.97	3.97	10.63	1.50×
IOUs	81.36% [80.70, 81.99]	2,501	109	3.39	3.58	6.83	1.77×
All three	93.50% [93.26, 93.73]	2,619	111	3.78	3.85	9.40	1.54×

Optimality depends strongly on debt structure. With uneven multi-payer bills, balances almost never contain zero-sum subgroups, so k − 1 transfers is already the minimum and greedy reaches it. With one-to-one IOUs, subgroups that cancel are common and greedy pairs across them. At ten people, greedy was optimal in only 25.2% [23.3, 27.1] of IOU ledgers. The largest excess in the whole study was three transfers. The pooled 93.5% depends entirely on the generator mix and should not be read as a general rate.

In every instance the greedy plan moved exactly the minimum total volume (half the sum of absolute balances). The direct plan needed on average 2.0 to 2.7 times as many

transfers as the minimum and moved 1.36 to 1.77 times the minimum volume. That is the price of never asking someone to pay a person they do not owe.

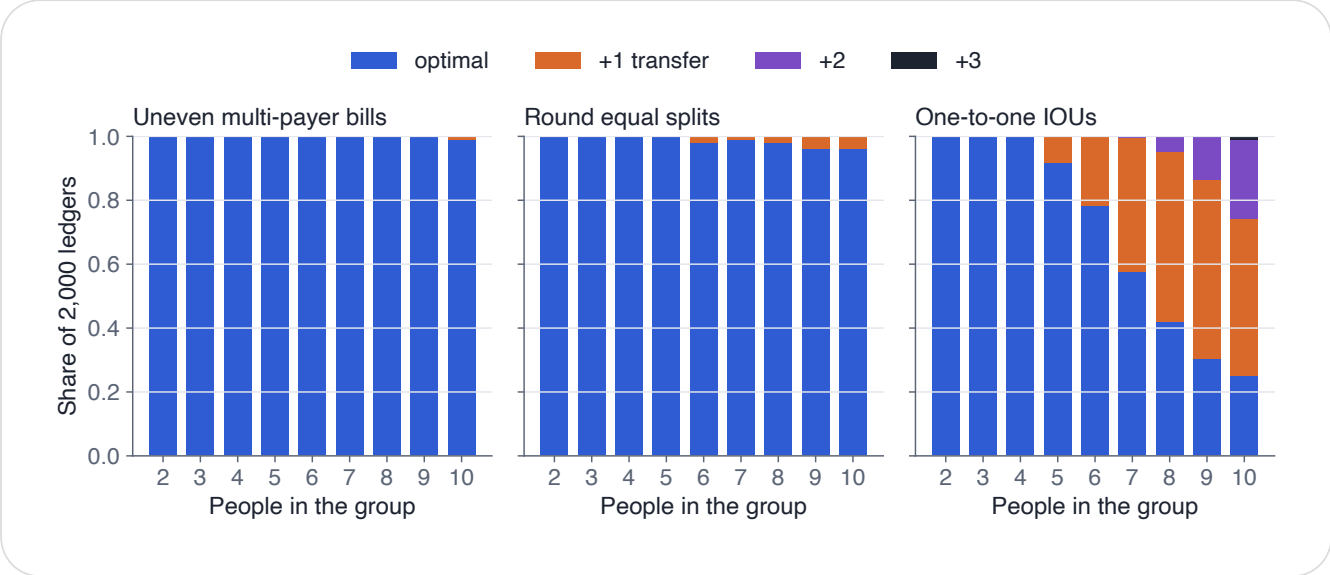


Figure 3. Distribution of the greedy plan's excess over the minimum number of transfers, by generator and group size (2,000 ledgers per bar).

Paying the simplified plan (E3). E3 generated 24,000 ledgers. In 23,665 of them (98.6%) the simplified plan had at least one rerouted row, meaning a row that does not match one of the payer's direct debts. Every row of the simplified plan was then paid, one at a time, under five strategies, and the final state was checked (Table 4, Figure 4).

Table 4. E3: paying every row of the simplified plan (24,000 ledgers). "Unsettled" means some balance is non-zero at the end. Brackets are Wilson 95% intervals, in per cent.

Strategy	Unsettled	Rows that moved nothing	Transfers	Other failure
Plain payment, clamped pair (no bridge)	90.0% [89.6, 90.3]	31,436 of 112,324	112,324	None
Plain payment, unclamped pair	0% [0, 0.016]	0	112,324	97.9% of ledgers left with phantom pair debts
Routed bridge (package)	38.6% [38.0, 39.2]	11,358 of 112,324	112,324	None
Bridge with fallback to the direct plan (bench only)	0% [0, 0.016]	0	196,454	None
Direct plan	0% [0, 0.016]	0	256,577	None

Without the bridge, a clamped pairwise ledger loses rerouted payments: 28.0% of rows moved nothing and 90.0% of groups ended unsettled. Removing the clamp seems to fix this, since net balances then reach zero in every ledger. However, 97.9% of ledgers are left with non-zero pairwise debts that cancel only in aggregate. A direct plan read from that state would ask people who are already square to pay one another in a cycle, and such reverse debts are exactly what the clamp exists to prevent.

The bridge removes most of the failures, but not all. 38.6% of groups still ended unsettled, and the rate rises steeply with group size: from 0% at three people to 41.3% at six and 86.3% at ten (87.0% for uneven bills, 85.5% for IOUs; Figure 4). The cause is the limitation that the package already documents and pins in a test. The greedy simplifier pairs debtors and creditors by amount alone, so it can ask someone to pay a creditor to whom no chain of real debts connects them, and then no flow exists for `decomposeRoutedPayment` to find. The measurements show that this is the common case for groups of eight or more, not a corner case.

A bench-only guard decomposes each row before recording it, and switches to the direct plan for the rest of the group as soon as a row cannot be fully covered. It settled every ledger. It fell back in 9,256 ledgers, exactly those the bridge alone left unsettled, and still

used 23.4% fewer transfers than the direct plan. No strategy ever broke conservation or flipped the sign of a balance.

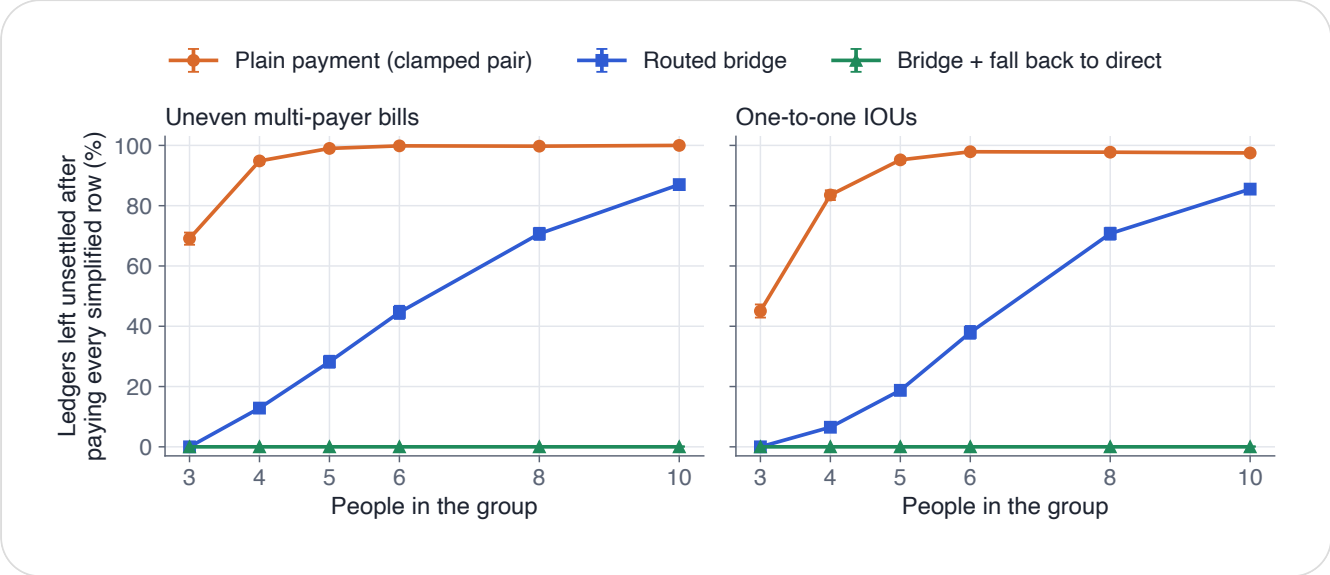


Figure 4. Share of groups left unsettled after every simplified row is paid, by recording strategy, generator and group size (2,000 ledgers per point, Wilson 95% intervals).

Staleness and tampering (E3). The covered edges are stored on the payment, so they can go stale, or be altered, before the ledger applies them. The campaign drew 20,000 IOU ledgers of 3 to 8 people, picked one simplified row, decomposed it, and then either mutated the edge list (inflating, redirecting, dropping or adding one edge) or left it intact while one of its debts was partly paid in the meantime. 18,841 seeds produced a non-empty decomposition and form the cases in Table 5.

Table 5. E3 tamper campaign by mutation. "Third party moved" means someone other than the payer and receiver changed balance. "Strict" is a bench-only rule that also requires every edge to be at most the debt it reduces at that moment.

Mutation	Cases	Rejected by validator	Third party moved (package)	Third party moved (strict)	A balance grew or flipped (package / strict)
Inflate an edge	3,813	3,754	0	0	0 / 0
Redirect an edge	3,774	2,656	51	0	36 / 0
Drop an edge	3,805	3,590	0	0	0 / 0
Add an edge	3,710	3,506	0	0	0 / 0
Stale (a debt partly paid since)	3,739	0	502	0	1,679 / 1,501
Total	18,841	13,506 (71.7%)	553 (2.9% [2.7, 3.2])	0	1,715 / 1,501

Money was conserved in every case. In 553 cases, however, a third party saw their balance change. 51 were redirected lists that still formed a balanced flow along an edge where no debt existed: the clamped reduction on that edge did nothing while the next edge was reduced. The other 502 were stale lists, in which one edge's debt had already shrunk and its reduction was clamped short. The strict rule brought third-party movement to zero in both categories.

A separate effect remains under both rules. In stale cases, the row's amount was computed before the intervening payment and can exceed what the payer now owes, so a balance grew or changed sign (1,679 cases under the package rule, 1,501 under the strict rule). This is a stale-plan effect, not an edge-list effect, and it reflects money that was really paid. The strict rule also has a cost: whether it accepts a payment depends on what was applied before it, so applying it at replay time would make the ledger order-dependent, which the current design avoids. Applying it once, when the payment is recorded, avoids that cost.

6.4 RQ4: cost

E5 times each operation on one seeded ledger per (n, m). Because the machine was heavily loaded, wall-clock medians were up to 62 times the CPU minimum, so wall time is not reported as a result. Table 6 gives process CPU time per call (user plus system, via `getrusage`), as the best of 21 samples of at least 20 ms each, after a 300 ms warm-up. Medians and wall times are in the artefact.

Table 6. E5: CPU time per call in µs (best of 21 samples), AOT-compiled, Apple M1 Pro under heavy shared load.

Group, expenses	Net balances	Direct plan	Simplified plan	Decompose all rows
3 people, 10	22	14	22	1.8
5 people, 100	335	194	315	6.3
10 people, 100	675	394	664	89
10 people, 1,000	6,434	3,901	7,264	67
20 people, 1,000	14,577	7,995	14,917	872

The balance and plan operations scale roughly linearly with the number of expenses and grow with the number of participants per expense (Figure 5). A group of five to ten people with 100 expenses needs well under a millisecond of CPU per recomputation; a 20-person group with 1,000 expenses needs about 15 ms. `distributeCents` on its own takes 0.8 to 7.2 µs for 3 to 20 people.

Net balances cost about 1.6 to 1.8 times as much as the direct plan in Table 6, although both use the same kernel. The reason is that `netBalances` calls `Expense.shares()` a second time to list every member, which repeats the `BigInt` division. Caching shares per expense would remove that cost. Because each figure comes from one ledger on a loaded machine, these numbers indicate orders of magnitude, not precise costs.



Figure 5. CPU time per call, best of 21 samples, for a group of ten people.

7 Discussion

7.1 What the numbers say

The case for integer minor units is usually made with `0.1 + 0.2`. These measurements support a sharper version. In a shared-expense ledger the main danger of floating point is not accumulated error, which stayed below 10^{-6} minor units even over 10,000 expenses. It is that a representation holding fractional cents must round late and independently, and independent rounding does not conserve. Allocating whole minor units at the moment of the split, with one rule, removes the problem at its source. It also makes results independent of input order, which doubles do not (E4).

The same experiments show that "exact" is not the whole story. Each of the three weaknesses below has a practical consequence and a cheap candidate fix.

7.2 Tie-break bias

A deterministic tie-break is what lets two devices agree to the cent, but a fixed one charges the same person every time (Section 6.2). Rotating ties by a hash of the expense

id keeps every exactness and determinism property, because the result is still a pure function of the inputs, and it removes almost all of the drift.

For a developer, the implication is that conservation tests alone cannot detect this bias; a test must compare each member's balance with the exact rational balance over long ledgers. Rotating the tie-break would change some existing splits by one minor unit, so an app that adopts it needs a migration note or a version flag on old expenses.

7.3 Routed-bridge settlement

The routed bridge is necessary, since without it most rerouted payments are lost, but it is not sufficient: 38.6% of groups stayed unsettled, rising to 86.3% at ten people. The results support the app's choice to use the direct plan for pair balances and to treat the simplified plan as an optional, explained view.

The practical rule is that an app should not offer a simplified row as payable unless the row decomposes fully into real debts at that moment. The guard that falls back to the direct plan settled every ledger while keeping 23.4% fewer transfers than the direct plan. A connectivity-aware simplifier, which never pairs people that no chain of debts connects, would address the cause rather than the symptom.

7.4 Tampering and staleness

Covered edges are data stored on a payment and applied later. The validator rejected most malformed lists and never let money be created or destroyed, but it allowed a third party's balance to move in 2.9% of cases, mostly because a list went stale between recording and application.

In a synced app, the implication depends on who can write edge lists. If any client can submit them, a crafted but balanced list could shift a bystander's balance, so validation against current debts belongs on a trusted server at record time. The strict per-edge check removed all third-party movement in this campaign. Run once at record time, it does not compromise replay order independence.

---

8 Threats to validity

8.1 Internal validity

The naive baseline is my own re-implementation, and a different naive implementation could fail differently. Rounding only at display time, or using a decimal type, would avoid

the round-each failures but not the display-sum failures. I tried to make the baseline representative by sharing every input with the integer run and by reporting three variants plus an exact rational control. The exact reference and the bench-only rules and guards are themselves code that has not been through the package's test suite, so a bug in them would bias the comparisons; the agreement of the exact control with the raw doubles at the 10^{-7} level is a partial check.

The E5 timings were taken on a heavily loaded shared machine. CPU-time minima are much more stable than wall time, but they are not equivalent to timings on an idle machine.

8.2 External validity

All workloads are synthetic. The amount distribution, split mix, payer mix and group sizes are plausible choices, not measurements of real groups, and Section 6.3 shows that conclusions about the simplifier depend heavily on debt structure. No production data was used. The results come from one implementation in one language, on one machine and one Dart SDK. The VM and JavaScript builds agreed on E4's digest, which is consistent with Dart's documented integer semantics below 2^{53}, but other runtimes were not tested.

8.3 Construct validity

"Unsettled" counts any non-zero final balance, however small. "Gap from exact" treats the exact rational share as the fair reference; other definitions of fairness, such as rounding in favour of the person who paid, are defensible. The number of transfers is one measure of plan quality, and users may care more about whom they have to pay than about how many payments there are. "Displayed" balances are modelled as rounding half away from zero to the minor unit.

8.4 Conclusion validity

Intervals assume that seeded ledgers behave as independent samples from the generator; they quantify sampling noise within the synthetic design, not uncertainty about real use. Pooled figures average over heterogeneous cells, so per-cell results (in the artefact) should be preferred where they differ. Zero-failure results are reported with upper bounds rather than as proof of absence.

9 Limitations

- **Property tests are not proofs.** The 153 tests and these experiments sample the input space with fixed seeds and no shrinking. The invariants (conservation, mirrored refunds, order independence) are simple enough to state in a proof assistant or to check exhaustively over small domains, which would be stronger evidence.
- **The simplifier is not optimal.** It was up to three transfers above the minimum here, and optimal in only a quarter of ten-person IOU groups. An exact subset dynamic program is cheap at these group sizes (2^{10} subsets for ten people).
- **The three fixes are not shipped.** Rotated ties, the fallback guard and the strict edge check exist only in the bench and have not been through the package's tests or the app.
- **No production data, and app flows not evaluated.** The report makes no claim about how real groups use the app or how often these failure modes occur in practice. The app's own settle-up sheet, which already sizes routed payments by the decomposable flow, was not measured.
- **Private source.** The package source is private. The benchmark code, raw results and file hashes are public, so others can audit the analysis and recompute every number, but cannot rerun the experiments.
- **Single developer.** The design, the code and this evaluation come from one person working with AI coding assistants. There has been no independent review, and the report has not been peer reviewed.
- **Other known gaps.** Currencies with exponent 4 are not supported; `SplitMethod.exact` trusts the caller to make the parts sum to the total; and `netBalances` computes shares twice (Section 6.4).

10 Conclusion

Holding money as integer minor units and routing every division through one deterministic rule kept the Bupples ledger core exact in every case tested. Across about four million synthetic expenses it created or lost no minor units and gave the same outputs for every input order, while a double implementation that rounds each share left about half of all splits off their bill.

Exactness is not the whole story. A fixed tie-break drifts against particular members, and rotating ties per expense removes most of that drift. Recording payments from the simplified plan needs both the routed bridge and a fallback to the direct plan. Routing data stored on a payment should be validated strictly, once, when the payment is recorded.

These results come from synthetic workloads on one implementation, and the three fixes have so far been evaluated only in the bench. Within those limits, they turn the package's informal claims into measured ones and identify what to change next.

Artefact availability

The artefact at /research/cent-exact-ledger/artifact/ contains the experiment package (`cent_exact_bench`), the raw per-ledger CSV files and JSON summaries behind every number in this report, the figure script, SHA-256 sums of the measured `bupples_ledger` 0.1.0 files, and a README with the exact commands, seeds, Dart version and dates. The package source is part of the private Bupples codebase and is not included, so the experiments can be rerun only with access to it. On 2 November 2026 a clean copy, with the package in place, reproduced the E6 summary byte for byte and the E4 platform digest exactly.

References

1. M. Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. Money pattern, catalogue entry. martinfowler.com/eaaCatalog/money.html
2. D. Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys 23(1):5–48, 1991. doi.org/10.1145/103162.103163
3. IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019). IEEE, 2019. doi.org/10.1109/IEEESTD.2019.8766229
4. W. Kahan. Pracniques: further remarks on reducing truncation errors. Communications of the ACM 8(1):40, 1965. doi.org/10.1145/363707.363723
5. International Organization for Standardization. ISO 4217: Codes for the representation of currencies. iso.org/iso-4217-currency-codes.html

6. Dart team. Number representation. Dart language documentation (native int is 64-bit; on the web, numbers are JavaScript doubles). dart.dev/resources/language/number-representation

7. M. L. Balinski and H. P. Young. Fair Representation: Meeting the Ideal of One Man, One Vote. 2nd edition, Brookings Institution Press, 2001 (1st edition Yale University Press, 1982). brookings.edu/books/fair-representation

8. T. Verhoeff. Settling multiple debts efficiently: an invitation to computing science. Informatics in Education 3(1):105–126, 2004. doi.org/10.15388/infedu.2004.08

9. C. Pătcaş and A. Bartha. Evolutionary solving of the debts' clearing problem. arXiv:1402.6556, 2014. arxiv.org/abs/1402.6556

10. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00), pages 268–279, 2000. doi.org/10.1145/351240.351266

11. D. R. MacIver, Z. Hatfield-Dodds and others. Hypothesis: a new approach to property-based testing. Journal of Open Source Software 4(43):1891, 2019. doi.org/10.21105/joss.01891

Cite and reuse

Authorship and AI assistance. Yousof Selim designed the product and the ledger rules and directed the work. Much of the implementation and parts of this report were produced with AI coding assistants (Claude, Codex) under his direction and review. All experiments are reproducible from the artifact. This is a technical report and has not been peer reviewed.

```
@techreport{selim2026cent,
  author      = {Yousof Selim},
  title       = {Cent-Exact by Construction: An Empirical Study of an In
  institution = {Sunway University (Lancaster University dual degree), M
  type        = {Technical report},
  number      = {v1.0},
  year        = {2026},
  url         = {https://yousofselim.com/research/cent-exact-ledger/}
}
```