# rounderr

an AI powered reverse mode rounding error interpreter in Python
design guide, Part I the engine, Part II the AI layer, Part III one machine and Bean

Eric Shi  CS 4999 with Dr. David Bindel

November 2026

**What the program does in one breath**

You type a formula (plain text, LaTeX, or a line by line tape). The program reads it back to you the way it understood it, asks for the numbers, runs the formula once forward while writing a tape, runs the tape once backward to get every line's weight $\lambda_i$, and then prints three kinds of numbers. **What actually happened** (computed value, 60 digit true value, actual error, and the first order prediction of that error from the measured slips). **What the error might be** (the worst case ceiling, the typical size, the problem's condition number $\kappa$, the algorithm's amplification $A$, and Bindel's backward error $\beta$). **Why and what to do** (which line carries the blame, which subtraction magnifies it, and rewrite candidates you can pick and rerun with the same numbers). Everything can be written out as a LaTeX report.

# Part I  The engine (`rounderr.py`)

## 1  The math it runs

The notation is the one from the whiteboard and from Bindel's FPTalks 2026 slides. The content is Linnainmaa's 1970 thesis and 1976 paper.

**Tape, slips, adjoint, three readings of one equation**

**Tape.** Every rounded step is one line $t_i = G_i(t_{<i}, x)$. Written as a system,

$$F(t, x) = t - G(t, x) = 0, \qquad y = w^T t, \quad w = e_n \text{ picks the last line.}$$

**Slips.** The machine does not compute $t_i$ but $\hat{t}_i$. The slip of line $i$ is the computed value minus the exact operation applied to the already rounded operands,

$$\rho_i = \hat{t}_i - G_i(\hat{t}, \hat{x}), \qquad |\rho_i| \le d_i \, |t_i| \, u,$$

which is Linnainmaa's local error $r_i$ (his equation 3). $u = 2^{-53}$ for double. $d_i = 1$ for an IEEE operation, 2 for a library function (1 ulp), 0 for a step that is exact.
**Linearize** $F(\hat{t}, \hat{x}) = \rho$ around the exact run,

$$F_t \, \Delta t + F_x \, \Delta x = \rho, \qquad \Delta y = w^T \Delta t \quad \Longrightarrow \quad \Delta y = \lambda^T \rho - \lambda^T F_x \, \Delta x, \qquad \lambda = F_t^{-T} w.$$

$F_t = I - G_t$ is unit lower triangular, so $F_t^T \lambda = w$ is a back substitution. That back substitution *is* the reverse sweep (Linnainmaa's Algorithm T).

**Three readings.**

| set | result | name |
|---|---|---|
| $\rho = 0$ | $\nabla_x y = -F_x^T \lambda = G_x^T \lambda$ | reverse mode gradient |
| $\Delta x = 0$ | $\Delta y = \lambda^T \rho$ | forward error (Linnainmaa, first order) |
| $\Delta y = 0$ | $\lambda^T F_x \, \Delta x = \lambda^T \rho, \;\; \|\Delta x\|_2 = \|\lambda^T \rho\| / \|g\|_2$ | backward error, $g = \nabla_x y$ |

**Numbers the program prints.** With $v_i = \lambda_i d_i t_i$ over the tape lines,

$$\text{ceiling} = u \sum_i |v_i|, \qquad \text{typical} = \tfrac{u}{\sqrt{3}} \, \|v\|_2, \qquad \text{predicted} = \sum_i \lambda_i \rho_i \;\; (\text{measured } \rho_i),$$

$$\kappa = \frac{\sum_j |g_j x_j|}{|y|}, \qquad A = \frac{\sum_i |v_i|}{|y|}, \qquad \beta_1 = \frac{\|v\|_1}{\|g\|_2 \, \|x\|_2}, \qquad \beta_2 = \frac{\|v\|_2}{\|g\|_2 \, \|x\|_2}.$$

$\kappa u$ is the relative error the *problem* forces on any algorithm. $Au$ is what *this algorithm* allows. The verdict rule is $A \le \max(10, n) \max(1, \kappa)$ means stable, otherwise unstable, where $n$ is the number of rounded steps (the allowance grows with $n$, as the $n\varepsilon$ bounds Bean proves for sums and Horner do). $\beta_1$ is the smallest relative input change (in units of $u$) that would explain the worst case output error. Small $\beta$ means backward stable.

**Worked numbers, the quadratic root $y = -b + \sqrt{b^2 - c}$ at $b = 1000$, $c = 1$**

| line | $t_i$ | $d_i$ | $\lambda_i$ | note |
|---|---|---|---|---|
| $t_2 = -b$ | $-1000$ | 0 | 1 | sign only |
| $t_3 = b \cdot b$ | $10^6$ | 0 | $5 \times 10^{-4}$ | small integers, exact |
| $t_4 = t_3 - c$ | 999999 | 0 | $5 \times 10^{-4}$ | small integers, exact |
| $t_5 = \sqrt{t_4}$ | 999.9995 | 1 | 1 | the only slip |
| $t_6 = t_2 + t_5$ | $-5.0000013 \times 10^{-4}$ | 0 | 1 | Sterbenz, exact |

Gradient $(\partial y/\partial b, \;\partial y/\partial c) = (5 \times 10^{-7}, \; -5 \times 10^{-4})$, so $\kappa = 2$. Ceiling $= u \cdot 999.9995 = 1.1 \times 10^{-13}$, which is $A = 2 \times 10^6$ in units of $u$ (charging every line once gives the September number $2000\,u$ absolute, $4 \times 10^6\,u$ relative). Actual error $-4.36 \times 10^{-14} = -7.9 \times 10^5 u$, and $\sum \lambda_i \rho_i$ predicts exactly that. $\beta_1 = 2000$. Verdict unstable, $A/\kappa = 10^6$. The program finds the rewrite $-c/(b + \sqrt{b^2 - c})$ by itself. Rerun with the same numbers it has $A = 2.5$ and actual error $-0.05\,u$.

## 2 How it is built, step by step

The file is one Python module, `rounderr.py`, about 1000 lines in nine numbered sections. Each step below names the section and says what to write.

### Step 1 Read the formula (section 1, the reader)

1. **Normalize.** Replace unicode (×, ÷, √, π, ²). If a backslash is present, expand LaTeX with a tiny scanner. `\frac{A}{B}` becomes `((A)/(B))` (recursive, so nesting works), `\sqrt{A}` becomes `sqrt(A)`, `\sin` becomes `sin`, `\left` and `\right` are dropped. Then `^` becomes `**`, braces become parentheses, `|x|` becomes `abs(x)`.

2. **Tokenize with Python's own tokenizer.** No hand written parser. Walk the tokens and insert `*` between a value and a value (`2x`, `4ac`, `(a)(b)`, `sin(x)cos(x)`). A multi letter name that is not a known function, not declared, and not of the form letter plus digits is split into letters (`ac` becomes `a*c`). `sin x` becomes `sin(x)`. `ln` becomes `log`. Every guess is pushed to a `notes` list.

3. **Tidy and echo.** `ast.parse` then `ast.unparse` gives a clean string. Print "I read it as" plus the notes, and ask for confirmation. Variables are the names that are never assigned, in order of first use.

4. **Scripts.** Lines with `=` are a tape you wrote by hand. Each left side becomes a name the later lines can use. The last line is the output.

**Why not sympy for the reading**

`sympy.parse_expr` rewrites $a/b$ as $a \cdot b^{-1}$ and reorders sums. That changes the tape, so it changes the rounding analysis. The `ast` module keeps the user's exact order and keeps subtraction and division as what they are. That is also why there is no `eval` anywhere.

### Step 2 Build the tape (section 2)

A `Line` holds `op`, the indices of its operands, and later its value, slip, $d$, partials and $\lambda$. Walk the AST once with a recursive `build(node)` that returns the index of the line it made.

| AST node | tape line |
|---|---|
| `Constant 0.1` | `const` line (storing 0.1 already rounds, so it is a line) |
| `Name b` | the input line made at the start |
| `BinOp +, -, *, /` | `add sub mul div` with two operand indices |
| `x ** 2` | `sq`, one multiply, one rounding |
| `x ** 3` up to `** 8` | a chain of multiplies, as a numerical program would write it |
| `x ** 0.5` | `sqrt` |
| `e ** x` | `exp` |
| `Call sqrt(...)` and friends | one line per call |
| `-x` | `neg`, exact |

### Step 3 Forward pass, values and slips (section 3)

For every line, in order,

1. compute the value the machine computes. At 53 bits this is the real machine (Python floats and the C library). At any other precision $p$, take the exact result and round it to $p$ significant bits

yourself with `round_to_bits` (ties to even). That gives correctly rounded single, half, bfloat16 or anything up to 53 bits with no extra library.

2. compute the *exact* result of the same operation on the same rounded operands. For $+ - \times \div$ use `fractions.Fraction`, because `Fraction(float)` is exact. For `sqrt`, `exp`, `sin`, ... use `mpmath` at 60 digits.

3. slip $\rho_i = \hat{t}_i - \text{exact}$, and the relative slip $\delta_i = \rho_i / \text{exact}$.

4. $d_i$ from the exactness rules (table below). A slip measured as exactly 0 is also noted, separately, as "measured exact".

5. the local partial derivatives at the rounded values, from the table below.

6. for add and sub, digits lost $= \log_{10}\left(\max(|a|, |b|)/|t_i|\right)$, and an absorption flag when one operand is under $10^{-3}$ of the other.

A second pass, `reference`, runs the whole tape in 60 digits from the *stored* inputs and the *exact* constants. Its last value is the true $y$, and actual error $= \hat{y} - y$.

| operation | $d_i$ | rule |
|---|---|---|
| `neg`, `abs` | 0 | sign only |
| `add`, `sub`, `mul`, `sq` on small integers | 0 | integer results below $2^{p/2}$ are exact |
| `add` or `sub` with a zero operand | 0 | adding zero |
| true subtraction with $y/2 \le x \le 2y$ | 0 | Sterbenz lemma |
| `mul` or `div` by $\pm 2^k$ | 0 | exponent shift only |
| `add sub mul div sq sqrt` otherwise | 1 | IEEE correctly rounded, $\lvert\delta\rvert \le u$ |
| `exp log sin cos tan pow` ... at 53 bits | 2 | C library, allowed 1 ulp (FPTaylor's convention) |
| the same at an emulated precision | 1 | the emulation rounds the 60 digit value correctly |
| `const` | 0 or 1 | 1 only when the decimal literal is not a binary fraction |

| op | $\partial/\partial a$ | $\partial/\partial b$ | op and derivative |
|---|---|---|---|
| $a + b$ | 1 | 1 | $\sqrt{a}$ $1/(2t)$ |
| $a - b$ | 1 | $-1$ | $e^a$ $t$ |
| $a \cdot b$ | $b$ | $a$ | $\ln a$ $1/a$ |
| $a/b$ | $1/b$ | $-a/b^2$ | $\sin a$ $\cos a$, $\cos a$ $-\sin a$ |
| $a \cdot a$ | $2a$ | | $\tan a$ $1 + t^2$, $\arctan a$ $1/(1 + a^2)$ |
| $a^b$ | $b\,a^{b-1}$ | $t \ln a$ | $\operatorname{expm1} a$ $t + 1$, $\operatorname{log1p} a$ $1/(1 + a)$ |

Here $t$ is the line's own computed value, which saves a function call.

## Step 4 Backward pass (section 4)

This is the whole of reverse mode. Six lines.

```
def backward(tape):
    for ln in tape.lines: ln.lam = 0.0
    tape.lines[tape.out].lam = 1.0                  # T1  start with 1 at the output
    for ln in reversed(tape.lines):                 # T6  walk the tape backwards
        if ln.op in ('input', 'const'): continue
        for j, dj in zip(ln.args, ln.partials):     # T5  pass the blame to each operand
            tape.lines[j].lam += ln.lam * dj        #     += handles a value used twice
```

Memory hook. Multiply along the path, add across paths, start with 1 at the output. The `+=` is what makes $x \cdot x$ and any reused value come out right. The `-matrix` option builds $L = F_t$ and $B = G_x$ explicitly, solves $L^T\lambda = w$ by back substitution and shows that it equals the sweep to the last bit.

### Step 5 The numbers (section 5)

`analyze` fills a `Result`. `y`, `y_true`, `actual`, `predicted` $= \sum \lambda_i \rho_i$, `ceiling`, `typical`, the gradient from the input lines, `kappa`, `amp` ($A$), `beta1`, `beta2`, the verdict, and two warnings. One fires when the ceiling passes 10% of $|y|$ (first order is meaningless there). The other fires when predicted and actual differ by more than 10% (second order effects). The agreement of predicted and actual is also a free correctness test of the whole backward pass, Langlois' CENA identity.

### Step 6 Why and what to do (section 6)

1. Rank lines by $|\lambda_i d_i t_i|$. For each of the top lines with at least 3% share, say its share and its weight $|\lambda_i t_i|/|y|$ (one $u$ of slip here becomes that many $u$ in $y$).
2. Search downstream from that line for the cancelling add or sub that loses the most digits. If one exists, name it. The subtraction itself is usually exact (Sterbenz), the damage is that it keeps the slip of its operands while throwing away the good digits.
3. Otherwise mention absorption, a library function, or an inexact constant.
4. If the verdict is unstable, or $A > 100$, try the rewrite patterns on the AST and offer the matches as numbered candidates. Picking one reruns the analysis with the same inputs and prints a comparison table.

| pattern | replacement |
|---|---|
| $-A + \sqrt{A^2 - R}$ or $\sqrt{A^2 - R} - A$ | $-R/(A + \sqrt{A^2 - R})$ |
| $A - \sqrt{A^2 - R}$ | $R/(A + \sqrt{A^2 - R})$ |
| $\sqrt{A^2 + R} - A$ | $R/(\sqrt{A^2 + R} + A)$ |
| $\sqrt{A} - \sqrt{B}$ | $(A - B)/(\sqrt{A} + \sqrt{B})$ |
| $1 - \cos X$ | $2\sin^2(X/2)$ |
| $e^X - 1$ | $\mathrm{expm1}(X)$ |
| $\ln(1 + X)$ | $\mathrm{log1p}(X)$ |
| $X^2 - Y^2$ | $(X - Y)(X + Y)$ |
| $1/X - 1/Y$ | $(Y - X)/(XY)$ |

Matching is structural. $A$ must appear as the same subtree in both places, checked with `ast.dump`. The replacement is spliced in with a `NodeTransformer` and printed back with `ast.unparse`.

### Step 7 Reports (section 7)

The text report is what you saw above. The LaTeX report has the same blocks in colored boxes, a `longtable` of the tape, and the formula typeset by a 60 line AST to LaTeX printer (`TexPrinter`) that knows precedence, `\frac`, `\sqrt` and the function names. `-pdf` runs `pdflatex` if it is installed.

### Step 8 Command line and interactive loop (section 9)

```
python rounderr.py                                          # interactive, with 10 examples
python rounderr.py "-b + sqrt(b**2 - c)" b=1000 c=1    # one shot
python rounderr.py "\frac{-b+\sqrt{b^2-4ac}}{2a}" a=1 b=1000 c=1 --precision single
python rounderr.py tape.txt b=1000 c=1 --matrix        # a file with t1 = ..., t2 = ...
python rounderr.py "(1 - cos(x))/x**2" x=1e-4 --compare --tex report.tex --pdf
python rounderr.py --selftest
```

Names given on the command line are kept whole, so `ac=3` makes `ac` a variable instead of `a*c`. Values may be `1e-4`, `3/7`, `pi/4` or `2**-20`.

## 3 Validation

Every row is one run of the program. $A$, $\kappa$ and $\beta_1$ are in units of $u$ relative to $y$. Predicted matched actual to the printed digits in every row.

| formula | inputs | bits | $A$ | $\kappa$ | actual/$u$ | verdict |
|---|---|---|---|---|---|---|
| $a-b$ | 1.5, 1.25 | 53 | 0 | 11 | 0 | every line exact |
| $-b+\sqrt{b^2-c}$ | $10^3$, 1 | 53 | $2\times10^6$ | 2 | $-7.9\times10^5$ | unstable |
| $-c/(b+\sqrt{b^2-c})$ | $10^3$, 1 | 53 | 2.5 | 2 | $-0.05$ | stable |
| $(1-\cos x)/x^2$ | $10^{-4}$ | 53 | $4\times10^8$ | $9\times10^{-9}$ | $-4.7\times10^7$ | unstable |
| $2\sin^2(x/2)/x^2$ | $10^{-4}$ | 53 | 7 | $2\times10^{-9}$ | 1.4 | stable |
| $e^x-1$ | $10^{-10}$ | 53 | $2\times10^{10}$ | 1 | $7.4\times10^8$ | unstable |
| expm1$(x)$ | $10^{-10}$ | 53 | 2 | 1 | $-0.3$ | stable |
| $\ln(1+x)$ | $10^{-10}$ | 53 | $10^{10}$ | 1 | $7.5\times10^8$ | unstable |
| log1p$(x)$ | $10^{-10}$ | 53 | 2 | 1 | 0.3 | stable |
| $x^2-y^2$ | 1.0000001, 1 | 53 | $5\times10^6$ | $2\times10^7$ | $-3.6\times10^5$ | stable, ill conditioned |
| $(x-y)(x+y)$ | same | 53 | 2 | $2\times10^7$ | 1.1 | stable |
| $0.1+0.2+0.3+0.4$ | | 53 | 2.9 | 0 | 0 | stable |
| Horner $(x-1)^3$ | $x=1.0001$ | 53 | $5\times10^{12}$ | $8\times10^{12}$ | $8\times10^{11}$ | stable, $\beta_1=0.55$ |
| $(-b+\sqrt{b^2-4ac})/2a$ | 1, $10^3$, 1 | 24 | $7\times10^5$ | 2 | $-1.2\times10^5$ | unstable |
| $-4ac/(b+\sqrt{b^2-4ac})/2a$ | same | 24 | 2.75 | 2 | 0.36 | stable |

Three things to notice. Sums, Horner and $x^2-y^2$ are backward stable even when their forward error is large, because $A$ stays near $\kappa$. The quadratic, $1-\cos x$, $e^x-1$ and $\ln(1+x)$ are the opposite, $\kappa$ is tiny and $A$ is huge, so a rewrite fixes them. And the first order prediction $\sum \lambda_i \rho_i$ reproduced the actual error in every row, which is the check that the sweep is right.

## 4 How to extend it

- **A new function** $f$. Add its name to `FUNCS`, its float call to `native`, its high precision call to `hp_func`, and its derivative to `partials`. Four lines. Its $d$ is already 2 at 53 bits.
- **A new rewrite.** Add one `if` to `rewrite_node` that recognizes the shape and returns the new node with two descriptions, plain and LaTeX.
- **A new precision.** Nothing to add, `-bits 8` already works because the emulation rounds the exact value.
- **Mixed precision.** Give `Line` a `bits` field and use it in `forward`. The ceiling formula does not change, only $u$ becomes $u_i$, which is Bindel's $D$ matrix.
- **Julia port.** `Line` becomes a `struct`, the tape a `Vector{Line}`, `Fraction` becomes `Rational{BigInt}`, mpmath becomes `BigFloat` with `setprecision(200)`, and the reader walks `Meta.parse` output instead of `ast`. The backward function is the same six lines.
- **Finding bad inputs** (Bindel's ongoing work). $\beta_1(x)$ is already computed. Its gradient needs one more forward and backward pass, then a few steps of gradient ascent on $x$ find the inputs where the algorithm is worst.

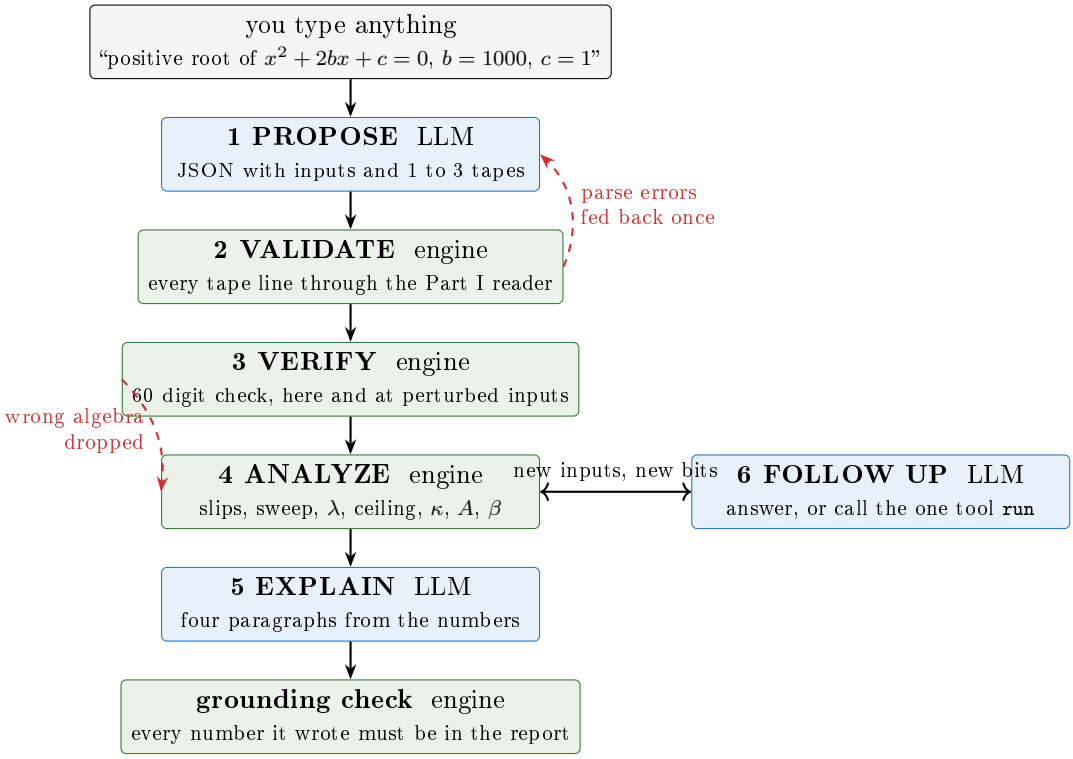
**Limits to keep in mind**

Straight line formulas only, no loops or branches (a loop unrolled into a script works). First order only, so the two warnings matter. The exactness rules are data dependent and are printed as notes, the "every line charged once" ceiling is the data independent number. Library functions are charged 1 ulp by convention, the measured slip shows what the library really did. The optional `-llm` switch only rewrites the finished report in words, it never touches a number.

# Part II The AI powered layer (`rounderr_ai.py`)

**The structure in one sentence**

An existing LLM does the three things it is good at, reading messy human input, proposing algebraically equal ways to compute the same thing, and putting numbers into words. The tape engine from Part I does the three things the LLM must never do, parsing into an exact tape, computing every number, and checking the LLM. **The LLM proposes, the engine disposes.**




## Division of labor

| | |
|---|---|
| the LLM | reads any phrasing, LaTeX or prose, picks out the inputs, writes tape 1 exactly as the user meant it, proposes equal rewrites it is sure of, and explains the finished numbers in plain words |
| the engine | parses every tape line with the Part I reader, evaluates all tapes in 60 digits to catch wrong algebra, runs the forward pass and the backward sweep, computes every number, and checks that the explanation only quotes numbers that exist |
| neither | the LLM never adds, multiplies or rounds anything, and the engine never guesses what the user meant |

### Existing LLM ideas and where each one is used

| idea | where | why it matters here |
|---|---|---|
| program aided LMs, tool use | step 1, the model writes a tape, the interpreter runs it | arithmetic by a language model is the one thing a rounding error tool cannot afford |
| structured output, a JSON contract | every LLM reply is one JSON object with fixed keys | the engine can validate it mechanically and feed errors back |
| few shot prompting | three worked examples of the contract inside the prompt | the tape language is small and unusual, examples teach it faster than rules |
| self refine | validation problems are returned to the model once | most JSON or tape slips are fixed on the second try |
| verification by execution | step 3, all tapes compared in 60 digits at the given and at perturbed inputs | an equal looking rewrite that is wrong algebra is dropped with numeric evidence |
| grounded generation plus a faithfulness check | step 5, the narrative is checked number by number against the report | an invented number in an explanation is worse than no explanation |
| ReAct style agent with one tool | step 6, answer or call `run(inputs, bits)` and then answer | lets the student ask "what if $b = 10^6$" or "single precision" in words |
| temperature zero | every call | the same question should give the same tape |

### The contract

```
{"reading": "y = -b + sqrt(b^2 - c)",
 "inputs":  {"b": "1000", "c": "1"},      <- copied as strings, the engine makes them
    exact
 "missing": [],                           <- input names the user forgot
 "tapes": [
   {"name": "textbook",
    "lines": ["t1 = b*b", "t2 = t1 - c", "t3 = sqrt(t2)", "y = -b + t3"],
    "why": "as written"},
   {"name": "conjugate",
    "lines": ["t1 = b*b", "t2 = t1 - c", "t3 = sqrt(t2)", "t4 = b + t3", "y = -c / t4"],
    "why": "multiply top and bottom by b + sqrt(b^2 - c)"}]}
```

Three prompts, one per role. **PROPOSE** carries the tape language (the allowed operations, one rounded operation per line, last line named `y`, keep the user's order in tape 1, never do arithmetic) and the three examples. **EXPLAIN** asks for four titled paragraphs, at most three sentences each, every number copied from the report. **AGENT** allows exactly two replies, `run` with new inputs or bits, or `answer`. The mock provider in the file obeys the same three roles, so the whole pipeline and its self test run offline.

### Guards, one per failure

| failure | guard |
|---|---|
| the model's rewrite is wrong algebra | step 3 evaluates every tape in 60 digits at the user's inputs and at three perturbed points, a mismatch above $10^{-30}$ relative drops the tape and prints both values |
| the model mistranscribed the user's formula | when the user text itself parses as a formula, tape 1 is checked against that formula the same way |
| the model invented a number in the explanation | the grounding check extracts every number from the narrative and looks for it (within 5 percent) in the engine report, misses are listed in red |
| a value is missing | the `missing` list, and command line `name=value` pairs override the model |
| the reply is not JSON | the first `{ ... }` block is extracted, one retry with the problems listed |
| the agent keeps calling the tool | three calls at most, then the comparison table is the answer |
| no API key | `-provider mock` runs the same pipeline with a rule based stand in |

### How to run

```
pip install anthropic mpmath
export ANTHROPIC_API_KEY=...
python rounderr_ai.py "positive root of x^2 + 2bx + c = 0 with b = 1000, c = 1" --tex ai.
    tex --pdf
python rounderr_ai.py "\frac{1-\cos x}{x^2} at x = 1e-4" --lang zh
python rounderr_ai.py "Horner for 2x^3 - 3x + 1" x=1.0001 --bits 24
python rounderr_ai.py "..." --provider openai --base-url http://localhost:11434/v1 --model
     llama3.1
python rounderr_ai.py "..." --provider mock          # no key, same pipeline
python rounderr_ai.py --selftest
```

After the report the program waits for follow up questions. "why is t5 to blame" is answered from the report. "what if b = 1e6" or "single precision" makes the model call `run`, the engine reruns, and the answer comes from the new report. Everything, including the follow ups, goes into the LaTeX report.

### What the self test checks

A deliberately wrong rewrite is dropped by the 60 digit check. The quadratic yields two tapes, both kept, $A = 2 \times 10^6$ and 2.5. The narrative passes the grounding check, and an invented number fails it. "single precision" reruns the engine at 24 bits, "b = 1e6" reruns with the new input. Plain formula text goes through the generic path and is verified against the user's own formula.

### How to evaluate it for the write up

The engine is an oracle, so grading is automatic. Take the fifteen formulas of the Part I validation table as a benchmark, each written in three styles (plain, LaTeX, prose). For each model (a small one, a large one, and the mock as a rule based baseline) measure three rates. Tape correctness, the share of tape 1 that pass verification against the user's formula. Rewrite discovery, the share of unstable cases where a proposed tape is kept and has $A$ under $\max(10, n)\max(1, \kappa)$. Grounding, the share of narratives with no loose number. Report the three rates and the median wall time per formula. That table is the AI half of the project, the Part I numbers are the numerical half.

**What this layer does not change**

Every number in a report still comes from the Part I engine. If the LLM is removed, `rounderr.py` still works on its own with its rule based reader and rewrite patterns. The LLM widens what can be typed and what can be asked, it does not touch the mathematics.

# Part III One machine (`rounderr_app.py`) and Bean (`rounderr_bean.py`)

**The whole pipeline in one window**

`python rounderr_app.py` opens a local page. Type on the left, press Analyze, read the tabs on the right. Nothing leaves the computer unless a model reader is switched on.

input → READ → TAPE → HASH → table lookup → matrix SWEEP → BEAN → DATA

| stage | what it does and where it lives |
|---|---|
| READ | the Part I reader, or a model that writes tapes (Part II). Numbers may sit in the text, as in "`... with b = 1000, c = 1`" |
| TAPE | one line per rounded step, shown as a paper strip with a blame bar per line |
| HASH | SHA 256 of the tape structure. Operations and wiring only, no numbers and no names, so $ab+c$ and $pq+r$ get the same key. The engine makes it, not the model, so it is reproducible |
| table | `rounderr_cache.json`. Per key it keeps the pattern of $L$, Bean's answer and a hit count. Model proposals are kept under a hash of the typed text, so the same text never calls the model twice |
| SWEEP | $L^T\lambda = w$ by back substitution. For random trials the tape is compiled once per key and every input runs at once, each $\lambda_i$ is a column of numbers (numpy) |
| BEAN | the tape is translated to a Bean program and type checked. Bean's bound holds for all inputs, the sweep holds at these inputs |
| DATA | every run is a row in the log, trials give distributions and the worst inputs found, exports are CSV, `.tex` and PDF |

### Developer tab

The window has a Developer tab for anyone working on the code. It lists the nine stages in run order. Each stage opens to three things. What the stage is designed to do. What it received and produced on the current run (the text after each reading step, the tape, the exact string that was hashed and the table entry, every line's operands, computed value, exact value, slip and $d$, every $\lambda$ update of the sweep in order, the verdict arithmetic with the numbers filled in, the Bean program and Bean's raw output, the model's JSON when a model was used). And the source of the functions that did it, read live from the files with `inspect`, with file names and line numbers. Below that are the file map, the local API routes and the raw JSON of the run. The same trace prints in a terminal with `python rounderr_app.py -trace "formula with x = 1"`.

### The fast sweep

`FastTape` walks the tape once with arrays in place of numbers. Forward gives the values, the exactness flags $d_i$ (Sterbenz, powers of two, small integers, written as array tests) and the local partials. Backward is the same six line loop, now on columns. It returns $A$, $\kappa$ and $\beta_1$ for every input. The self test checks it against the full engine on all examples (largest gap $2 \times 10^{-16}$). On the quadratic, 1000 inputs take about 2 milliseconds. The slow part of the full engine is the exact slips and the 60 digit reference, which the fast path skips, so the worst input found is rerun through the full engine to confirm it.

### Bean

Translation rules. $a + b$ and $a - b$ become `add` and `sub`. $a \cdot b$ becomes `mul`, or `dmul` when one side is reused. $a/b$ becomes `div` inside a `case`. A negation is folded into the next add or sub. A value used more than once must be discrete and may only multiply. Constants become inputs. `sqrt`, `cos` and the rest are not expressible. To compare, an output error $\Delta y$ is turned into the smallest relative change of the inputs Bean may perturb, $|\Delta y| / \sum_j |g_j x_j|$.

| formula | Bean bound ($\varepsilon$) | adjoint ceiling ($u$) | measured ($u$) | share used |
|---|---|---|---|---|
| sum of 4 | 3 | 1.9 | 0.25 | 13% |
| 2 by 2 determinant, nearly singular | 1.5 | 0.25 | 0.036 | 2% |
| Horner cubic near a triple root | 6 | 0.625 | 0.1 | 3% |
| sum of 50, worst of 200 random inputs | 49 | 12.9 | 1.78 | 7% |
| dot product of 20, worst of 200 | 20 | 9.01 | 2 | 16% |
| Horner degree 20, worst of 200 | 40 | 14.1 | 2.04 | 10% |

Bean typed 11 of 18 test formulas. In 608 engine runs the measured backward error never left Bean's bound. Bean could not take 7 formulas (no `sqrt` or `cos`, or a reused value outside a multiply), and the engine calls 3 of those stable, so a rejection is not a proof of instability. The comparison also fixed the engine. A fixed cutoff of 10 called a 49 step sum unstable while Bean proves it stable, so the cutoff is now $\max(10, n)$.

### Run it

```
pip install mpmath numpy            (anthropic too, for the Claude reader)
python rounderr_app.py              opens the window
python rounderr_app.py --bean path/to/bean.exe
python rounderr_app.py --selftest   6 checks
```

Bean is optional. Build it from https://github.com/Athena-Types/Bean with `opam install -deps-only .` and `dune build`. Without it the Bean tab still shows the translated program.

## References


- S. Linnainmaa, *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*, MS thesis, University of Helsinki, 1970.
- S. Linnainmaa, Taylor expansion of the accumulated rounding error, *BIT* 16 (1976) 146–160.
- S. Linnainmaa, Error linearization as an effective tool for experimental analysis of the numerical stability of algorithms, *BIT* 23 (1983) 346–359.
- D. Bindel, Adjoints for roundoff sensitivity analysis, FPTalks, 6 Aug 2026, https://www.cs.cornell.edu/~bindel/present/2026-08-fptalks.pdf.
- N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, 2002 (chapters 1, 3, 4, 5).

- P. Langlois, Automatic linear correction of rounding errors (CENA), *BIT* 41 (2001) 515–539.
- A. Solovyev et al., Rigorous estimation of floating point round off errors with symbolic Taylor expansions (FPTaylor), *ACM TOPLAS* 2018.
- H. Menon et al., ADAPT, automated algorithmic differentiation for floating point precision tuning, SC 2018.
- A. Das et al., Scalable yet rigorous floating point error analysis (SATIRE), SC 2020.
- A. Kellison, *Type based approaches to rounding error analysis*, PhD dissertation, 2025, arXiv 2501.14598 (NumFuzz and Bean, with Bindel and Hsu).