Which side of the line?

A question so simple that computers get it wrong, and what it takes to prove they do

Munawar Kazmi • munawarkazmi.com

A plain-language guide to **exact-predicates**, an open demonstration of a bug class and its fix.

Code, corpus, and every figure: github.com/munawarkazmi/exact-predicates

---

The simplest question in geometry

Draw two points on a page and rule a line through them. Now mark a third point somewhere.

Is that third point to the left of the line, to the right of it, or exactly on it?

You can answer instantly by looking. A computer answers by arithmetic: it subtracts some coordinates, multiplies them together, and checks whether the result comes out positive, negative or zero. Positive means left, negative means right, zero means exactly on the line.

This calculation sits underneath an enormous amount of software. Every time a program works out whether two things collide, whether a point is inside a shape, which route goes around an obstacle, or how to stitch a map together, something very like it is running, usually thousands of times a second.

And it can be wrong.

Almost right is a different thing from right

Computers do not store most numbers exactly. They store extremely close approximations, accurate to roughly sixteen decimal digits, and that is normally more than anyone needs.

The trouble starts when the answer you want is very close to zero.

If the third point is nearly, but not quite, on the line, then the true answer is a tiny number. The computer, meanwhile, has arrived at that tiny number by multiplying large numbers together and subtracting the results. Each of those steps rounds slightly. If the rounding errors are larger than the answer itself, then the sign that comes out, the left or right or on-the-line, is decided by accumulated noise rather than by geometry.

The result is not a crash and not an obviously silly answer. It is a confident, wrong, single word. Left, when the truth is right.

> This is not a hypothetical. It happened in a sibling project: a robot route planner kept freezing stale reasoning into its paths because two priorities that were mathematically equal ended up stored a hair apart, so the comparison that should have said “these are the same” said “this one is slightly bigger”. The routes it returned were occasionally not the best ones, and nothing in the output explained why.

This repository is that lesson turned into its own artefact.

A construction that leaves no room for argument

To show the problem is real, you need inputs where the true answer is definite and the computer still gets it wrong. Vague, borderline cases prove nothing.

The corpus uses a construction based on Fibonacci numbers, the sequence where each number is the sum of the previous two. Take three points built from consecutive Fibonacci numbers, and a classical identity guarantees that the true answer to the which-side question is exactly plus or minus one. Not nearly zero. Not ambiguous. A definite, provable, non-zero answer.

But those Fibonacci numbers are enormous, around a billion billion. To reach its verdict the computer multiplies them together, producing intermediate values with something like thirty-six digits, then subtracts two such values to find a difference of exactly one.

Rounding at that scale is vastly larger than one. The true answer is real, definite, and completely invisible to the arithmetic being used to find it.

The fix, and the boundary around it

The repair is the same idea as in the robot planner: stop using approximations. Within a stated range of coordinate sizes, the calculation is performed in whole numbers using a register wide enough to hold every intermediate value without rounding anything, ever. No approximation enters, so no approximation can corrupt the sign.

That works only inside a range, and this is where the project does something slightly unusual. The range is not merely documented, it is enforced. Every calculation checks its inputs against the limit and refuses, throwing an error, before performing any arithmetic at all.

> A documented limit that is not checked is a claim that is true on the page and unenforced in the code. Feed it a number one step too large and it will quietly overflow and hand back a confidently wrong answer, while the documentation still says “exact”. The single most important test in this repository feeds coordinates one past the limit, in every position, and insists the refusal fires.

The project also states plainly what it is not. It is a focused demonstration of one bug class and one guarantee, not a general-purpose geometry library. For exact answers over arbitrary numbers rather than the restricted range here, the README points readers to the established libraries built for that, on the grounds that reimplementing them would risk producing something that looks right and is subtly wrong, which is precisely the failure this project exists to expose.

Proving the danger, not just the fix

Most projects that fix a correctness bug test the fix. This one tests both halves, and the difference matters.

The repository commits 657 adversarial cases: 400 for the which-side question, 154 for a related test about whether a point falls inside a circle through three others, 43 about whether two segments touch, and 60 configurations for building the outline around a cloud of points.

On every single one of those cases, the automated tests assert two things:

- that the ordinary approximate calculation gives the **wrong** answer, and

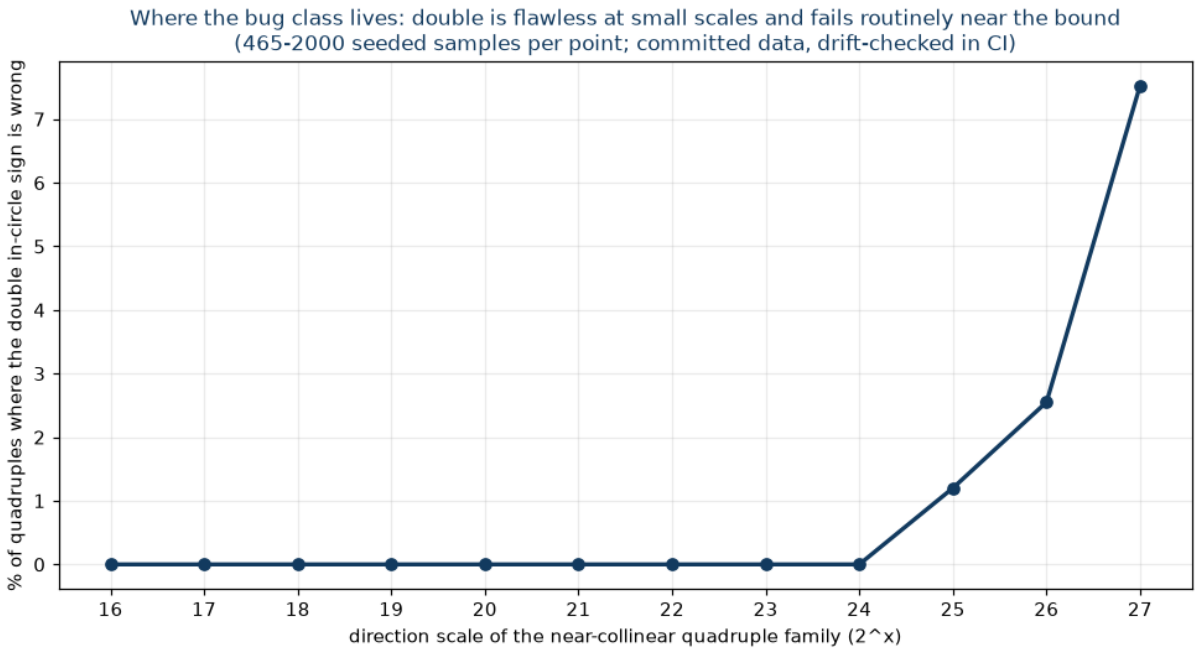

Figure 1: The honest map of the danger. Below a coordinate scale of roughly sixteen million, ordinary approximate arithmetic gets this family of cases right every time. Past that, it starts failing, and the failure rate climbs steeply. A corpus built at small scales would have "proved" a danger that does not exist there.

- that the exact calculation gives the **right** one.

Both directions, case by case, on every change to the code. If a future version of the compiler or the hardware quietly made the approximate version correct on some case, the test would fail and someone would have to look at it. The danger is demonstrated rather than asserted, which is a stronger claim than "we fixed a bug we say existed".

Separately, an independent implementation written in a language with unlimited-precision whole numbers, and with no range restriction at all, re-checks every committed case: 1,554 of them in range, plus 4 deliberately outside the range where the main implementation refuses to answer, showing that the limit is a restriction of this implementation rather than of the problem. Zero disagreements.

Honesty about where the danger is not

Here is the part that most impressed me about the design, and it cuts against the project's own interest.

It would be easy to build a corpus of scary-looking failures at small coordinate scales and declare the problem universal. So the repository measured where the failure actually begins.

The README says it directly: well-spread points cannot embarrass an approximate calculation, and pretending otherwise would be building a strawman. The bug class is real, and it lives in a specific place, and the project documents where that place is rather than implying it is everywhere.

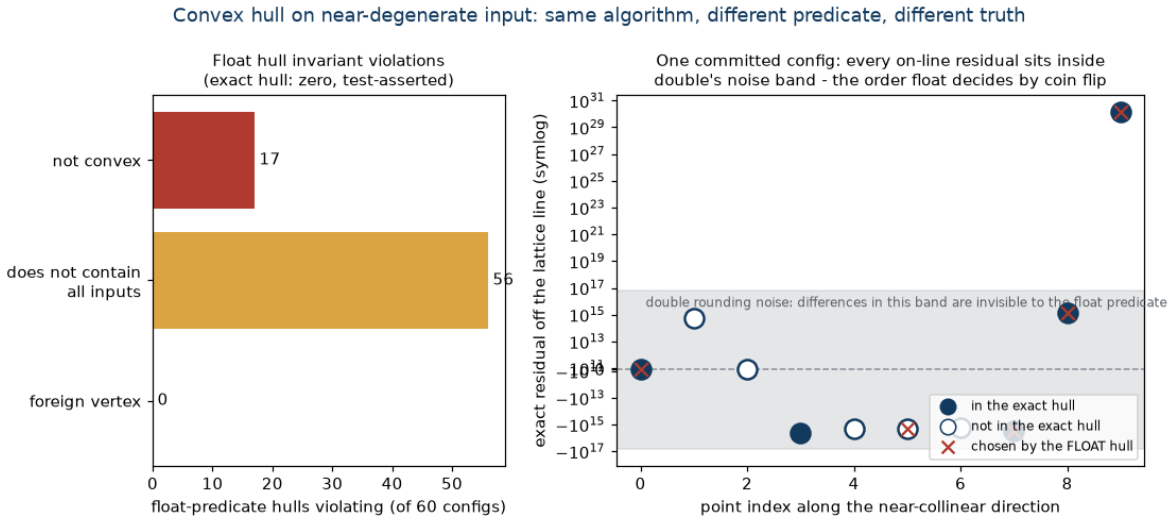

Figure 2: Same algorithm, same inputs, different arithmetic. On the left, how the approximate version fails across 60 committed configurations. On the right, one of those configurations: the shaded band is the range within which the approximate calculation cannot distinguish points at all, so the ordering it produces inside that band is effectively arbitrary.

What it looks like when it breaks

The abstract version of this problem is a wrong sign. The concrete version is a shape that is not a shape.

Take a cloud of points and ask for the outline that encloses them all, the shape you would get by stretching an elastic band around the outside. This is a standard, well-understood algorithm, and it calls the which-side question repeatedly.

Run that identical algorithm on 60 committed near-degenerate point clouds, changing nothing but which version of the which-side test it uses. With the exact version, the answer is a correct outline every time. With the approximate version, 17 of the resulting outlines are not convex at all, and 56 fail to actually contain all of the points they are supposed to enclose.

An outline that does not contain its own points is the kind of result that produces a bug report three modules downstream, in code that is entirely correct.

What exactness costs

It is slower, and the project measures rather than hand-waves that. On the machine used, the which-side test takes about 50 nanoseconds exactly, against about 28 nanoseconds approximately: roughly 1.8 times the cost.

That is the whole trade, stated in one line. For most software, paying 1.8 times for a calculation that cannot silently corrupt everything downstream is an easy decision. For code running that calculation millions of times per second in a tight loop, it might not be, and the number is published so the decision can be made on evidence.

Why it matters

Bugs like this are unusually unpleasant, not because they are common but because of how they present. There is no crash to catch, no error message to search for, no obviously wrong number in a log. There is just an answer that is occasionally, quietly, incorrect, in a routine so basic that nobody thinks to suspect it.

The valuable move is not simply fixing it. It is committing the evidence that the problem was real, in a form that keeps re-proving itself long after everyone has forgotten the original bug.

Every adversarial case, every figure, the independent oracle, and the timing data are public, and regenerate exactly from committed code. This project grew out of a real bug in a sibling repository, a robot route planner that froze stale reasoning into its paths for exactly this reason.
Repository: github.com/munawarkazmi/exact-predicates