NAPA — A Working Engineer's Simulator

*a **AI** commentary on the package*

NAPA is a mixed-signal, sampled-data simulator built by an analog IC designer for analog and digital signal-processing engineers who need to reason precisely about systems that mix continuous (*R*, real-valued) and discrete (*I*, integer-valued) domains in the same netlist. It is not a general-purpose circuit simulator in the SPICE sense — it works at the behavioral/functional level, modeling filters, sigma-delta modulators, samplers, converters, and logic as a graph of typed nodes, each carrying an explicit domain signature ($R \to R$, $I \to I$, $R \to I$, $I \to R$), or adapting to its inputs when the node is "chameleonic" (gain, sum, delay, integrator, mux). That domain typing is the single most distinctive design decision in the language: it forces a mixed-signal description to stay honest about where a signal actually lives, catching a whole class of category errors — treating a digital code as an analog level, or vice versa — before a single sample is computed.

The architecture is equally distinctive, and deliberately unfashionable by today's standards: NAPA is not an interpreter. A `.nap` netlist is compiled — through a macro-expansion pass, then through the NAPA compiler proper — into a plain C source file, which is then handed to a real C compiler (MinGW64/GCC on Windows) and linked into a native executable. Running the simulation means running that executable. This is, in effect, a small domain-specific language with its own dedicated code generator, built decades before "compile netlists to native code" became a familiar idea in other tool ecosystems. The payoff is speed and transparency: the generated C is meant to be read, and a sigma-delta modulator running for millions of samples pays no interpreter overhead. The cost is that the toolchain — a macro preprocessor, a C compiler, a symbolic-math engine (Maxima, for deriving closed-form cell equations before they are discretized), and gnuplot for the output — has to be present and correctly wired together, which is a real dependency burden compared to a self-contained interpreter.

The cell library reflects a working engineer's priorities rather than an academic one's: on the order of 480 `.net` cells organized by family — classical filters, integrators in several topologies, PWM/PWL sources, sigma-delta variants (DSD, CDSD, ISD, ASD), data-weighted averaging, comparators, logic primitives — many of them documenting, in comments, the analytical derivation of the underlying differential equation before

giving its sampled-data implementation. That habit of showing the math next to the code is unusually disciplined for a tool this old, and it is genuinely useful: you can audit *why* a cell behaves the way it does, not just trust that it does.

Having spent a long session auditing the C engine itself — roughly twenty-one modules implementing the compiler's lexer, parser, code generator, and runtime support, on top of the macro preprocessor, plot front-end, and RTF documentation viewer — I can speak concretely to its engineering quality rather than just its feature list. The core compiler builds clean under `-std=c2x -ansi -Wall -Wextra -Wshadow -Wmissing-prototypes -Wredundant-decls -Wunreachable-code -Werror`, which is a stricter bar than most 25-to-30-year-old C codebases clear without extensive rework. For a codebase whose oldest comments date to 1997 and whose newest target is 64-bit Windows via MinGW, that is a genuinely good outcome, and it says something about how carefully the author has maintained it across three decades of incremental extension rather than rewrite.

The rough edges are the ones you would expect from that history rather than from carelessness: DOS-era naming conventions (`NapaDos`, `MacDos`, two-letter module names) that read as dated even though the code underneath is current; a distribution model built around a single Windows machine with a fixed drive-letter convention baked into batch scripts, icons, and generated banners rather than a portable installer; and documentation — a quick-reference card, a primer, a full reference manual, a teaser deck — that is thorough and precise but assumes the reader is already fluent in the domain, which is exactly the right choice for its actual audience and the wrong one for a newcomer hoping to be onboarded gently.

Taken as a whole, NAPA reads as a tool built by someone who needed it to be *correct* and *fast* far more than he needed it to be fashionable or portable, and who has kept it honest about that trade-off for a very long time. It is not trying to be a general platform; it is trying to be the right instrument for a specific, demanding kind of work, and on the evidence of the code itself, it succeeds at that on its own terms.