Patodia Equation Developer Guide

Patodia Equation Solver Version 4.0

Panna Lal Patodia
Patodia Infotech Private Limited
11 November 2026

This guide explains how to build, integrate, test and extend Patodia Equation Solver. It covers the command-line executable, native DLL, desktop host and the shared numerical engine. Use it alongside the public header and the source snapshot identified below.

Version 4.0 adds the workflow scripting interface. Sections 46–48 document execution, host integration and result handling. Earlier build identifiers and release records refer to the 3.0.1 baseline.

The retained Version 3.0.1 reference covers standalone services and the corresponding MSIX release gates. In addition to the retained algebraic, exact-integer, ODE, BVP, PDE, integral and quadrature contracts, it documents differentiation, tabular analysis, interpolation, limits, isolated Compute Functions, high-precision special functions, exact number theory and linear Diophantine families.

Document basis	Verified reference
Source package	Patodia_Equation_Solver_3_0_1_Complete.zip
Source identity	Version 3.0.1; numerical engine build 3.0.1.1
Build identification	Record the hash of the actual archive and deployed binaries; older archive hashes do not identify this edition.
Public interface	C ABI v1; route multiple result schemas by problem_type
Desktop host	Public 3.0.1; unchanged compatible binary file version 3.0.1.0
Numerical update	Mixed PDE boundaries, adaptive accuracy, eigen/transport/coupled integral/sweep and quadrature modes

Audience and use

Application developers can begin with Sections 2–9 and 32. Engine maintainers should read Sections 10–14 and 26–31. Sections 33–35 cover existing Desktop integration, tests and deployment; Sections 36–44 add the new standalone services, 113-bit backend and MSIX gates.

This is a pre-release review edition. Historical validation records retain their scope; they are not fresh Windows acceptance results. The long-value viewer limitation and the reported Windows quadrature endpoint-cancellation case remain open. The guide grants no distribution rights and does not replace the licence agreement.

Contents

1 Product architecture and source map
2 First integration and a reproducible model
3 Build the engine DLL and desktop
4 Equation language and model contracts
5 Command line options and process integration
6 JSON results precision and status
7 Native C and C plus plus integration
8 Python DLL integration
9 Host integration and lifetime rules
10 Numerical engine architecture
11 Exact integer search and factor recognition
12 Mathematical modulo and residue families
13 Interval certification and proof scope
14 Desktop host and appearance controls
15 Testing and regression design
16 Packaging licensing and deployment
17 Troubleshooting
18 Source references and extension checklist
19 Exact quadratic and large integer contracts
20 Initial value ODE integration
21 Automatic singular IVP continuation
22 Two point boundary value problems
23 PDE and integral equation integration
24 Result routing and licensing acceptance
25 Current source map and release validation
26 Boundary aware field algorithms and accuracy contracts
27 Regular Sturm Liouville implementation
28 Linear transport and coupled Volterra systems
29 Parameter sweep execution and JSON batches
30 Standalone quadrature parsing and expression evaluation
31 Quadrature transforms tail checks and acceptance
32 Mode aware result routing for host applications
33 Desktop panels numerical exports and snapshots
34 Regression design and platform acceptance
35 Build identity packaging and release gates

36 Standalone mode dispatch and isolated grammars

37 Expression and tabular differentiation implementation

38 Interpolation algorithms and CSV contracts

39 Numerical limits and evidence semantics

40 Compute Functions expression engine and explorer

41 Boost Multiprecision and precision boundaries

42 Compute number theory and exact integer outputs

43 Exact linear Diophantine classification

44 Windows MSIX assembly and release verification

45 Quadrature ranges singularity analysis and Desktop tools

46 Workflow execution and service dispatch

47 CLI and native DLL workflow integration

48 Workflow budgets and regression checks

Code conventions

Commands use Windows Command Prompt syntax unless labelled Linux. Run commands from the extracted source root, except where a change of directory is shown. The caret at the end of a Windows command line is its continuation character. Code examples use US number style and UTF-8 text. Source paths are relative to the extracted Patodia_Equation_Solver_Source directory.

A standalone model contains equations or service directives. A workflow script starts with workflow and can run multiple service models. Examples with different models must be run separately. Command-line options are tokens, and DLL options are arrays of tokens; neither interface accepts an arbitrary shell command as an option string.

1 Product architecture and source map

The numerical implementation is shared by two compute entry points: equation_solver.exe and equation_solver.dll. The Windows desktop program, PatodiaEquationSolver.exe, runs the executable as a separate worker process. It does not call eq_solve directly. This distinction matters for failure isolation, stopping work and DLL entitlements.

After statement and number-style normalisation and licensing checks, the dispatcher recognizes standalone compute/func, diff/differentiate, interp/interpolate, limit/lim and quad/quadrature models before handing remaining models to the existing PDE/eigen/integral/sweep, BVP, IVP or algebraic paths. Standalone modules have their own parsers, output schemas and validation contracts. Keep mode dispatch explicit to avoid changing the shared expression language unintentionally.

Source location	Responsibility
engine/src/main.cpp	CLI entry, Unicode Windows arguments, JSON output selection and precision dispatch
engine/src/library.cpp	C ABI, option allowlist, result ownership and per-call streams
engine/equation_solver.h	Public ABI declarations and memory contract
engine/src/engine.hpp	Shared includes, serialization helpers and engine assembly
engine/src/common.hpp	Numeric primitives, precision support and internal control state
engine/src/equations.hpp	Expressions, System, domains, options and search orchestration
engine/src/equations.tpp	Parser and numerical implementation, run control and reporting
engine/src/integer_search.hpp	Checked polynomial integer search and finite-domain propagation
engine/src/integer_factors.hpp	Products of two affine factors and exact rational recovery
engine/src/modulo_search.hpp	Integer remainder evaluation and residue search
engine/src/interval_proof.hpp	Native-first interval proof orchestration
engine/src/ibex_backend.hpp	External IBEX process and result translation
desktop/main.cpp and model.hpp	wxWidgets interface, worker management, projects and exports
engine/src/field_v3.hpp	Current boundary-aware PDE/eigen/integral algorithms, refinement and sweeps

Source location	Responsibility
engine/src/quadrature.hpp	Known-function definite/improper integration and scaled expression fallback
desktop/field_plot.hpp; desktop/studies_panel.hpp	Maps, contours, animation, convergence, eigen and sweep displays

The engine headers form a template implementation, with Engine<Real> selecting the arithmetic type. Some headers are included inside that class. In particular, integer_search.hpp includes integer_factors.hpp, which uses types and helpers declared in the surrounding class. Treat include placement as part of the implementation structure when refactoring.

Choose the integration boundary

Use the CLI when process isolation, a hard external timeout or a language-neutral interface is the priority. Use the DLL when an authorized application needs direct native calls and can manage native lifetime and deployment requirements. Build on the desktop host when adding interactive model editing or display behavior. Keep numerical semantics in the engine so CLI, DLL and desktop calculations remain consistent.

A developer should be able to reproduce a reported issue with a small model and explicit options before editing an algorithm. Preserve that reproducer as a test, then check the same input through both compute entry points.

2 First integration and a reproducible model

Extract every file in the Windows runtime ZIP to a normal folder. Keep bundled runtime DLLs beside the executable and library, and do not launch from inside the ZIP. The runtime package supplies ready-to-run binaries. Source build scripts create engine/bin and dist; if those folders are absent in a source-only archive, build them first or run the following examples against the separately extracted matching Windows runtime.

Save the following as factor_model.txt in the extracted source root. Explicit multiplication and an explicit variable order make the intended model clear.

```
var a, b
pos int a, b

a*(a+b) = 123456787

dist\equation_solver.exe --file factor_model.txt --check --json
dist\equation_solver.exe --file factor_model.txt ^
  --seconds 2 --max-solutions 100 --json
```

The syntax check returns status syntax_valid and performs no root search. A successful solve returns the following three pairs. The input needs no user-supplied starting point or explicit bound for this recognized factor structure.

a	b	Verification
1	123456786	1 × 123456787 = 123456787
31	3982446	31 × 3982477 = 123456787
961	127506	961 × 128467 = 123456787

The exact factor path enumerates divisors and checks positivity. Its reason text identifies exact affine factor enumeration. The general JSON search_complete field remains false because that field is reserved for the certification mechanism for nonconstant models; do not infer that this exact enumeration was a heuristic merely from that one field.

Check extended precision independently

```
dist\equation_solver.exe --eq "x^2=2; x>=0" ^
  --extend --seconds 2 --json
```

The expected root is a positive approximation to sqrt(2). Its coordinate, tolerance and residual are JSON strings. Retain them as decimal strings, or use a decimal type with sufficient precision. A small residual tests the entered equation; it does not prove that every displayed coordinate digit is correct.

For a minimal development baseline, record the executable hash, model bytes, option tokens, process exit code, stdout, stderr and parsed result. Keep the original response even when your application also produces a friendly table. It is the most useful evidence when a failure depends on bounds, precision or a resource limit.

3 Build the engine DLL and desktop

The engine requires a 64-bit GCC toolchain with C++17, true __float128 support and libquadmath. Include engine/third_party for the vendored Boost.Multiprecision headers used by exact integer algorithms. The Windows release uses MinGW-w64 GCC 13.2 POSIX. MSVC can call the DLL through its C ABI but cannot compile this implementation unchanged.

Windows engine build

From a Command Prompt with the matching x64 compiler on PATH:

```
g++ --version
g++ -print-file-name=libquadmath-0.dll
engine\build_windows.cmd
```

The batch file builds engine/bin/equation_solver.exe and equation_solver.dll, writes the MinGW import library libequation_solver.dll.a, and copies compiler runtime DLLs when found. A print-file-name result that is only the bare DLL name indicates it was not located. Verify the files rather than assuming the copy succeeded.

To rebuild the engine and refresh the existing desktop runtime together:

```
rebuild_engine_windows.cmd
```

rebuild_engine_windows.cmd refreshes engine/bin outputs into dist without rebuilding the GUI. Baseline product version is 3.0.1; the corrected engine EXE and DLL have file/product version 3.0.1.1. The compatible unchanged Desktop binary has file version 3.0.1.0. Rebuild the Desktop separately when its source changes and record both identities.

CMake alternatives

Configure only the engine when wxWidgets is not required:

```
cmake -S engine -B build-engine -G "MinGW Makefiles" ^
  -DCMAKE_BUILD_TYPE=Release
cmake --build build-engine --parallel 2
ctest --test-dir build-engine --output-on-failure
```

The engine CMake project checks quadmath and builds executable/shared-library targets, but its outputs are not a complete distribution. In this snapshot engine/CMakeLists.txt does not add src/version.rc to the engine targets. The supplied Windows batch build does compile and link CLI/DLL version resources. Add equivalent resource handling before using a CMake engine build as an installer payload; do not assume a resource-free binary satisfies file-version guards.

For a complete desktop rebuild, install matching wxWidgets 3.2.x sources, set WXWIN to their root and run build_desktop_windows.cmd. The supplied batch build uses static wxWidgets core, base and StyledTextCtrl libraries. The top-level CMake project also requires wxWidgets; optional font integration tests are enabled with PATODIA_ENABLE_GUI_TESTS=ON.

Linux development builds

The shared engine can be validated on Linux with GCC, quadmath and OpenSSL development libraries. This does not make the Windows delivery a Linux desktop release.

```
g++ -std=c++17 -O2 -DNDEBUG -pthread \
  -Iengine/third_party engine/src/main.cpp -lquadmath -lcrypto -o equation_solver

g++ -std=c++17 -O2 -DNDEBUG -pthread -fPIC -shared \
  -Iengine/third_party engine/src/library.cpp -lquadmath -lcrypto \
  -o libequation_solver.so
```

The platform-specific crypto dependencies come from license verification. On Windows the engine links BCrypt, Crypt32, Advapi32 and Shell32 instead of an OpenSSL DLL. Preserve the licence implementation when making a development build.

4 Equation language and model contracts

General algebraic models contain equations, inclusive conditions and declarations. Newlines or semicolons separate statements; comments are supported. Names are case-sensitive and explicit multiplication is recommended. Numerical modes instead start with ode, bvp, pde, integral, eigen or quadrature and have separate directive allowlists. Do not mix mode headers or inject algebraic bound overrides into numerical models.

```
var x, y
constant total = 5
int x
bound x = [0, 5]
bound y = [0, 5]
x+y = total
x*y = 6
```

The declared output order is x,y. The model has the two feasible pairs (2,3) and (3,2). var affects ordering rather than variable type. All listed variable names must occur in equations or conditions and cannot be constants. Additional unlisted variables follow alphabetically.

Construct	Meaning
int a,b	Unrestricted integer variables
pos int a,b	Positive integers; zero excluded
nonneg int a	Integers including zero and positive values
neg int a; nonpos int b	Negative integers; nonpositive integers respectively

Construct	Meaning
binary selected	Integer restricted to 0 or 1
constant R=10	Fixed name excluded from the unknowns
bound x=[L,U]	Hard inclusive bounds from constant expressions
x>=0; x<=10	Separate inclusive conditions

Repeated type declarations intersect their domains. Conflicts, duplicate file bounds or unknown bounded names are errors. Explicit CLI bounds override file bounds, and a named CLI bound overrides a global CLI bound. Declaration restrictions still apply, so a bound cannot turn a binary variable into an unrestricted integer.

Expressions and units

Use +, -, *, /, ^ or ** and parentheses. Division is real division even for declared integer variables. Powers associate right to left; -x^2 means -(x^2). Implicit multiplication is supported, but explicit multiplication is preferable in generated model text. Write x/(2*y) when the entire product belongs in the denominator.

Functions include trigonometric and hyperbolic functions, exp, ln/log, log10, sqrt, cbrt, abs, min, max, pow, atan2, hypot, log1p and expm1. Angles are radians. pi and #pi denote pi; #e denotes Euler's number, while bare e is a variable. Use real cbrt for a negative cube root.

The engine does not assign physical units. Scale equations deliberately and verify the original model in the same units as its coefficients. Tolerance is an absolute residual threshold; conditions use positive-part violation residuals. Strict inequalities and chained bounds such as 0<=x<=10 are rejected. Write two conditions or one bound declaration.

Number style

US style uses decimal points and commas between function arguments or bound endpoints. EU style uses decimal commas and semicolons for those separators. Use separate integer declarations in EU models. JSON remains decimal-dot in every style. Do not add thousands separators. Keep number-style conversion separate from mathematical changes to the model.

5 Command line options and process integration

Prefer --file PATH or --file - in automated callers. Existing positional paths are recognized, but a nonexistent positional path is interpreted as equation text. Explicit --file produces a clearer file-open failure. The stdin form also avoids embedding a large or confidential model in a command line visible to other process-inspection tools.

Option	Default in CLI engine	Purpose
--seconds S	20	Approximate allowance; 0 removes the time deadline
--starts N	12000	Multistart budget across stages
--iterations N	300	Local iteration ceiling per start

Option	Default in CLI engine	Purpose
--max-solutions N	500	Returned point limit
--seed N	42	Random seed for reproducibility
--tol T	1e-9 normally	Final residual threshold; extended default 1e-24
--radius R	10	Initial automatic radius
--growth G	10	Radius multiplier
--stages N	4	Expanding stages for automatic domains
--merge T	1e-6	Relative coordinate merging for real roots
--integer-neighbors N	8	Additional nearby integer assignments per start

For ordinary algebraic discovery, --extend selects hybrid double discovery and binary128 refinement; --quad-search requests binary128 discovery. Numerical IVP/BVP/field modes use their selected scalar type, while quadrature uses binary128 guard arithmetic even in standard mode. Root bounds and --tol do not replace numerical model domains, tolerance directives or refinement controls.

The desktop has its own settings defaults and passes them explicitly. Its default two-second allowance and 30000 starts must not be mistaken for CLI defaults. Likewise, --seconds 0 retains finite start and other resource limits; it does not mean an exhaustive unbounded search.

Python subprocess pattern

```
import json
import subprocess
from pathlib import Path

exe = Path("dist/equation_solver.exe").resolve()
model = "var a,b\npos int a,b\na*(a+b)=123456787"
try:
    run = subprocess.run(
        [str(exe), "--file", "-", "--json",
         "--seconds", "2", "--max-solutions", "100"],
        input=model, encoding="utf-8", capture_output=True,
        timeout=15, check=False)
except subprocess.TimeoutExpired as exc:
    raise RuntimeError("Worker exceeded host deadline") from exc

if not run.stdout.strip():
    raise RuntimeError(run.stderr or "No JSON response")
data = json.loads(run.stdout)
if run.returncode not in (0, 2):
    raise RuntimeError(data.get("error", run.stderr))
print(data["status"], len(data.get("solutions", [])))
```

Pass an argument array with shell=False, the subprocess default. Capture stderr separately and handle invalid or absent JSON, including loader failures before the engine starts. A host timeout terminates the worker and loses its final response. Do not invent a no-solution result from that interruption. The example does not preserve partial roots and does not implement a process-tree supervisor for optional external backends.

6 JSON results precision and status

Dispatch on schema and problem_type. Retained families are equation_solver.result.v1, equation_solver.ode_result.v1 and equation_solver.bvp_result.v1. PDE/integral/eigen use equation_solver.field_result.v1; parameter sweeps use equation_solver.sweep_result.v1 with nested runs; quadrature uses equation_solver.quadrature_result.v1 and a scalar value. Syntax checks and errors can have fewer fields. Section 32 provides a mode-aware routing example.

Field	Interpretation
status	solutions_found, no_solution_found, syntax_valid, error or another documented response state
variables	Ordered name and integer flag records
solutions[].values	Decimal strings in variable order
solutions[].maximum_residual	Decimal-string maximum original residual
solutions[].found_seconds	Acceptance time or null, not necessarily row order
solutions[].certificate_id	Proof association or null
precision and search_arithmetic	Output/refinement precision and discovery arithmetic
reason	Search stopping or specialized-search explanation
search_complete and scope	Certification-oriented completeness and its finite domain
certification	Enclosures, coverage, unresolved regions and backend calls

A nonconstant numerical result has top-level feasible=true when a point was found. feasible=false means no point was returned; it is not an infeasibility proof. A constant_satisfied result has no root vectors, so an empty solutions array alone does not imply failure. syntax_valid also has no roots because no search was requested.

Return and exit codes

Code	Contract
0	Successful solve or satisfied constant system; also successful syntax check
1	Input, option or engine error
2	No solution found, unsatisfied constant system, or a completed empty certification domain
3	Compute licence or entitlement failure
-1	eq_result_status called with a null result handle

The DLL normally returns an error result rather than a null handle. A null handle signals that a result could not be returned, for example allocation failure. Standalone CLI activation and metadata commands are separate paths and may emit text rather than solver JSON. Always retain their exit status and stderr.

Preserve precision and evidence

Preserve numerical and exact-integer decimal strings. Binary128 offers about 34 significant decimal digits of arithmetic; an error estimate, conditioning and input accuracy determine how many are justified. Quadrature reports binary128_guard in standard mode. Decimal parsing preserves exported digits but cannot increase their mathematical accuracy.

found_seconds records when a new distinct point first passed final acceptance. A later improved duplicate does not reset that time, and coordinate sorting may put later-discovered roots before earlier ones. Compare timestamps directly. Save the model, options, result and build identity together to make performance or root-count comparisons meaningful.

7 Native C and C plus plus integration

Use engine/equation_solver.h as the authoritative declaration. The DLL exposes a C calling convention and opaque eq_result handles; callers do not share C++ containers, exceptions or allocator ownership across the boundary. On Windows the process must be x64 and dependent DLLs must be discoverable.

```
#include "equation_solver.h"
#include <iostream>
#include <memory>
#include <string>

int main() {
    if (eq_abi_version() != 1) return 1;
    const char* options[] = {
        "--seconds", "2", "--max-solutions", "100"
    };
    using Result = std::unique_ptr<eq_result,
                                   decltype(&eq_result_free)>;
    Result result(eq_solve(
        "var a,b; pos int a,b; a*(a+b)=123456787",
        4, options), &eq_result_free);
    if (!result) return 1;
    int status = eq_result_status(result.get());
    const char* text = eq_result_json(result.get());
    std::string json = text ? text : "";
    const char* err = eq_result_error(result.get());
    if (status != 0 && status != 2)
        std::cerr << (err ? err : "Solver error") << '\n';
    std::cout << json << '\n';
    return status;
}
```

The option count is the number of tokens, not the number of named switches. The model string is passed separately. Copy any borrowed result string before freeing its handle. The unique_ptr deleter above guarantees eq_result_free is used even if later host code throws.

Link from MinGW or MSVC

Save the example as guide_example.cpp in the source root:

```
g++ -std=c++17 -Iengine guide_example.cpp ^
  -Lengine\bin -lequation_solver ^
  -o engine\bin\guide_example.exe
engine\bin\guide_example.exe
```

The supplied .dll.a is a MinGW import library. To create an MSVC import library, use a 64-bit Visual Studio developer prompt and the supplied export definition:

```
lib /def:engine\bin\equation_solver.def /machine:x64 ^
  /out:engine\bin\equation_solver.lib
cl /EHsc /Iengine guide_example.cpp ^
  /link /LIBPATH:engine\bin equation_solver.lib ^
  /OUT:engine\bin\guide_example.exe
```

These commands build a client of the existing DLL, not the GCC binary128 engine with MSVC. Keep the DLL and its runtime dependencies next to the resulting client, or use a controlled loader configuration. Match import declarations and the library release; do not hand-edit calling conventions to resolve link errors.

ABI surface

The ten exported functions are eq_abi_version, eq_has_binary128, eq_machine_fingerprint, eq_license_info, eq_solve, eq_result_status, eq_result_json, eq_result_report, eq_result_error and eq_result_free. The last three string accessors return borrowed UTF-8 strings. The fingerprint and licence-info functions use caller-supplied buffers and return required capacity including the NUL terminator, or -1 on failure.

8 Python DLL integration

Use ctypes.CDLL for the C calling convention. Declare every signature before the first call; otherwise ctypes may interpret a 64-bit pointer as a default integer and truncate it. The following complete example accepts the library path as its only argument. It can also be used with a native Linux .so for development validation.

```
import ctypes as C
import json
import os
import sys
from decimal import Decimal
from pathlib import Path

path = Path(sys.argv[1]).resolve()
search = None
if os.name == "nt":
    search = os.add_dll_directory(str(path.parent))
lib = C.CDLL(str(path))
lib.eq_abi_version.argtypes = []
lib.eq_abi_version.restype = C.c_uint
assert lib.eq_abi_version() == 1
lib.eq_solve.argtypes = [C.c_char_p, C.c_int,
                         C.POINTER(C.c_char_p)]
lib.eq_solve.restype = C.c_void_p
for name in ("eq_result_json", "eq_result_error"):
    fn = getattr(lib, name)
    fn.argtypes = [C.c_void_p]
    fn.restype = C.c_char_p
lib.eq_result_status.argtypes = [C.c_void_p]
lib.eq_result_status.restype = C.c_int
lib.eq_result_free.argtypes = [C.c_void_p]
lib.eq_result_free.restype = None
```

Continue in the same Python file after the declarations above.

```
model = b"var a,b; pos int a,b; a*(a+b)=123456787"
opts = [b"--seconds", b"2", b"--max-solutions", b"100"]
args = (C.c_char_p * len(opts))(*opts)
handle = lib.eq_solve(model, len(opts), args)
if not handle:
    raise MemoryError("No solver result handle")
try:
    status = lib.eq_result_status(handle)
    raw = lib.eq_result_json(handle)
    data = json.loads(raw.decode("utf-8") if raw else "{}")
    if status not in (0, 2):
        error = lib.eq_result_error(handle)
        raise RuntimeError(error.decode("utf-8") if error else
                           "Solver error")
    names = [v["name"] for v in data.get("variables", [])]
    rows = [dict(zip(names, map(Decimal, r["values"])))
            for r in data.get("solutions", [])]
    print(rows)
finally:
    lib.eq_result_free(handle)
    if search is not None:
        search.close()
```

Run it from the source root after saving it as guide_dll.py:

```
python guide_dll.py engine\bin\equation_solver.dll
```

The search-directory handle remains alive while the DLL is in use. A load failure can still mean a missing transitive dependency rather than a missing equation_solver.dll. Use a 64-bit Python interpreter and inspect all bundled dependencies. For non-ASCII model text, use text.encode("utf-8") instead of an ASCII bytes literal.

9 Host integration and lifetime rules

The public API is intentionally small. eq_solve is synchronous; it returns when its solve or error path finishes. Run it on a worker thread if your application must keep an interactive interface responsive. Independent calls can run concurrently and own separate results, but a result must not be freed while another thread is reading it.

The v1 ABI exposes no cancellation handle, streaming callback or progress notification. Internal solver_control state is an implementation detail, not an exported extension point. Use finite --seconds allowances for cooperative work limits. If a strict host deadline or forced cancellation is required, isolate the CLI in a supervised process rather than trying to terminate a thread executing DLL code.

Option validation

The DLL allows documented search, precision, number-style, bound, check and IBEX/certification controls. It rejects --file, positional model fragments, --eq, --int, --json, help, self-test and progress flags. Model declarations belong in the model string. JSON is obtained through eq_result_json, not by adding --json to DLL options.

Validate options in your wrapper, but retain the DLL's own allowlist. Do not let a user-supplied option value become a host shell fragment or executable path without application-level review. An IBEX path intentionally launches an external backend and should be a configured application resource.

Metadata and entitlement checks

Query eq_has_binary128 to inspect compiled capability, then inspect the active licence separately. A positive capability flag does not grant permission to use extended precision or DLL solving. For eq_license_info, pass a null path to select the default licence/trial lookup.

For metadata buffers, first call with a null buffer and capacity zero. If the returned size is positive, allocate that many bytes, call again and verify the second size fits. Allow a bounded retry if the required capacity changed. A return of -1 is failure, not an empty JSON document. Decode only a successfully filled UTF-8 buffer.

Other languages and application services

Host	Integration choice
C sharp or another managed runtime	PInvoke with cdecl, UTF-8 model bytes, explicit pointer-sized result handles and deterministic freeing
PHP	64-bit PHP with FFI enabled, or CLI subprocess isolation
Julia	ccall declarations matching the public C header
Web service	Bounded worker processes or authorized DLL workers with concurrency limits

The package supplies C, C++, Python, PHP and Julia examples under engine/examples. A managed-language binding should copy strings while the result is alive and release only through eq_result_free, preferably with a safe-handle abstraction. Do not call free, delete or Marshal.FreeHGlobal on DLL-owned strings.

Choose the numeric parsing locale at host startup and avoid changing process-wide locale while concurrent solves run. Use per-call --style US or EU for the model notation. Set practical model-size, concurrency, output and backend-file retention limits in a service wrapper; these are application policies rather than new solver guarantees.

10 Numerical engine architecture

System::parse, configureBounds and raw_residual describe the algebraic path. Earlier numerical dispatch routes to runQuadrature, runField, runBvp or runOde rather than building one universal algebraic System. field_v3.hpp and quadrature.hpp are included inside Engine<Real> after legacy field primitives; include placement is part of their type/helper dependencies.

The solve orchestration first tries modulo search, then exact integer search, then the specialized univariate polynomial path when eligible, followed by general numerical discovery. The polynomial branch has additional eligibility conditions; for example, the orchestration checks whether an onRoot callback is installed. These are implementation paths, not user-selected promises about every model of that broad mathematical type.

Continuous and mixed integer search

General discovery combines automatic search stages, deterministic low-discrepancy seeds and seeded random restarts. Bounds and inferred restrictions constrain the search. The local method uses damped least-squares restoration. Its least-squares helper applies Householder QR to an augmented system, avoiding explicit formation of J transpose J.

Affine seeding moves some starts toward affine equations as a hint. It is not a replacement of the model by a smaller system. Coordinate slices similarly restrict a search while retaining all original equations. Deflation helps avoid already found vectors during discovery; final polishing verifies the original residual without deflation.

For mixed integer systems, the numerical path considers nearby integer assignments and holds selected integer coordinates fixed during polishing. Final acceptance enforces declarations, bounds, finite evaluations and the requested original residual threshold. The exact all-integer paths in Sections 11 and 12 are separate accelerators.

Precision modes

Mode	Discovery	Final checking
Normal	Primarily double	Normal-mode acceptance at requested tolerance
--extend	Double candidate discovery	Binary128 polishing and original expressions
--extend --quad-search	Binary128 search	Binary128 polishing and verification

The polynomial safeguard path and standard-mode quadrature use internal binary128 arithmetic. Recognised arbitrary-precision integer routes preserve much larger exact values and eligible literal bounds; they are not limited to 2^53-1. The generic floating/mixed-integer and modulo paths retain their own exact-conversion limits. Never impose one global coordinate range on every result family.

Hybrid mode is useful for ordinary searches where faster discovery is adequate. Use full binary128 search when discovery needs narrow intervals or values not represented adequately in double. It can be much slower. Neither mode turns a small residual into a proof of completeness or a fixed number of correct coordinate digits.

Safe algorithm changes

Preserve final checking against the original system. A coarse discovery threshold, a transformed equation or a seed hint must not become the sole acceptance test. Keep budgets and duplicate handling consistent, and test the effects of seed changes and merge tolerance when roots are close together. Record improved root counts separately from run time; neither metric alone establishes a better solver.

11 Exact integer search and factor recognition

solveInteger accepts eligible all-integer polynomial models. It uses checked 64-bit coefficients with __int128 intermediate checks, rather than rounded floating coefficients. Integer polynomial parsing limits include 32 variables, powers through 16 and at most 512 stored terms. Decimal and scientific literals deliberately leave this exact polynomial path; a mathematically integral literal such as 12.0 is not the same input token as 12 for path selection.

The affine factor update calls solveIntegerFactors on original polynomial equations before the older variable-elimination pass. This placement fixes the reported case a*c=N with c=a+b: eliminating a first would hide the original product inside $c^2-bc=N$.

Recognize and recover candidate assignments

Equation form	Recognized structure
a*b=N	Two direct integer factors
a*(b+1)=N	Shifted factor
a*(a+b)=N	Factors sharing a variable
(a+2)*(b+3)=N	Two translated affine factors
(2*a+1)*(3*b+2)=N	Nonunit coefficients and integrality checks
a^2+a*b=N	Expanded common-variable factor
a*b+3*a+2*b+6=N	Expanded shifted-variable product

For F(x)G(x)=N, the shortcut enumerates signed divisors d and assigns F(x)=d, G(x)=N/d. It combines these with the model's other affine equalities and uses checked rational Gaussian elimination. The path is used only if those equalities determine every coordinate. A nonintegral recovered coordinate is rejected, rather than rounded.

Candidates must satisfy all original polynomial equations exactly, declared signs or binary restrictions, explicit hard bounds and the existing raw-residual acceptance test. Multiple factor representations do not authorize extra roots: the complete original system remains the acceptance condition.

For a*(a+b)=123456787, candidate a=d implies b=123456787/d−d. Positivity retains exactly (1,123456786), (31,3982446) and (961,127506). For a*(b+1)=123456787 there are five positive pairs because only the divisor giving b=0 is excluded.

Limits and fallback

This is not arbitrary symbolic factorization. Recognition inspects affine product nodes and recovers selected expanded common or shifted factors. Zero products, systems with undetermined coordinates, excessive divisors, oversized values, arithmetic overflow or preprocessing limits retain the earlier exact/numerical fallback. Under the current implementation, preprocessing also observes a speculative two-second/operation guard. A fallback must never be described as an infeasibility proof.

If the new shortcut cannot finish, the older integer path can eliminate eligible unbounded variables with unit linear coefficients, build divisor or explicit finite domains, filter modular residues and construct exact support relations. Small relations solve a coordinate linearly or quadratically instead of scanning every tuple. Propagation and conservative intervals reduce the search. Final reconstruction still checks the original equations.

When extending this code, preserve variable and equation order independence, signed-divisor handling, original-model verification and checked arithmetic. A useful new recognizer should either establish its finite search domain or decline the path; it must not silently treat an automatic numerical box as a global integer bound.

12 Mathematical modulo and residue families

The % operator is an integer mathematical remainder. For a positive modulus m, its result is between zero and m−1, including for a negative numerator. Thus (-4)%3 equals 2. This differs from the sign behavior of C++ remainder; wrapper implementations must not replace it with their language's native operator without matching the definition.

```
pos int x, y
x % 5 = 1
y % 6 = 2
(y-x) % 3 = 2
(x+y) % 9 = 5
(x*y) % 7 = 6
```

The pure congruence model has 18 representatives over reported periods x=315 and y=126. A representative is a class member, not the only solution in that class. Adding suitable integer multiples of the periods preserves the equations. For positive representatives in their positive period rectangle, nonnegative shifts generate the corresponding positive family.

To request individual solutions in a box, add explicit bounds for every variable:

```
bound x = [1, 100]
bound y = [1, 100]
```

This box contains (6,8), (51,80), (66,2) and (81,68). Automatic period inference is limited to pure forms P%m=r or r=P%m. General modulo expressions, nested modulo or combinations with ordinary inequalities require explicit finite bounds for all variables.

Syntax and implementation restrictions

All model variables must be integer when % occurs. Numerators support integer constants, variables, addition, subtraction, multiplication and constant integer powers through 64. Division and transcendental functions in numerators are rejected. The modulus must be a positive constant integer expression no larger than 1000000000. A constant remainder outside its valid range is unsatisfiable.

Multiplication, division and modulo share precedence and associate left to right. Write (x+y)%9 when the sum is intended as the numerator. x+y%9 means x+(y%9). Modular multiplication and powers are evaluated with integer modular arithmetic, avoiding floating rounding of intermediate products.

Current discrete limits include 32 variables, up to 20001 candidate integers per coordinate and inferred periods no larger than 20000. Larger pure periods require explicit finite bounds. Exact integer coordinate conversion remains limited to magnitude 2^53−1 even when extended precision is requested.

The modulo-specific reason reports representative periods or bounded-search termination. Preserve that message in an integration. The nonconstant top-level search_complete field remains certification-oriented and does not by itself distinguish a fully enumerated residue rectangle from a stopped heuristic search. If a time or solution limit interrupts modulo enumeration, do not present the returned subset as all classes.

13 Interval certification and proof scope

Ordinary numerical discovery finds approximate points. Optional --certify mode adds existence, uniqueness and remaining-domain coverage through an externally installed IBEX 2.9.1 executable. IBEX is not bundled in the supplied runtime, and its path must be configured explicitly. Legacy --ibex-assisted discovery and native-first --certify are different modes.

The following is a template for an installation with IBEX at the stated path. Change that path to the actual installed executable before running it.

```
dist\equation_solver.exe --eq "x^2=2" ^
  --bounds -2 2 --extend --seconds 10 ^
  --discovery-seconds 0.25 --certify ^
  --ibex "D:\ibex\ibexsolve.exe" ^
  --ibex-output proof_runs --json
```

What the certificate establishes

A supported certificate concerns real roots of the literal equations within the reported finite box. Its existence enclosure contains a true root and its uniqueness region limits the number of roots there. The printed point remains an approximation. A root certificate does not establish all roots were found; completeness also needs coverage of the remaining domain.

certification.coverage_complete indicates that coverage is exhausted with certified enclosures for its roots. Top-level search_complete additionally requires a returned point approximation associated with each registered certificate. A point limit can therefore leave search_complete false even when enclosure coverage is complete. global_complete remains false; no statement about every point of R to the n is implied.

Read unresolved_boxes, identity_ambiguous, detail, certificate records and certification.calls. In native-first proof mode the top-level legacy ibex object is not the complete proof log. An empty final remaining-domain IBEX result can mean no additional roots after certified neighborhoods were excluded; previously registered certificates still belong to the combined result.

Supported and unsupported models

The adapter supports a square continuous equality system with finite, positive-width intervals and supported smooth arithmetic/functions. Integer variables, user inequalities, underdetermined or overdetermined systems, zero-width domains and several nonsmooth or special operations are unsupported for certification. Singular roots, boundaries or discontinuities can leave unresolved regions even for a supported model.

Proof export preserves decimal literals through exact small-integer arithmetic expressions instead of claiming the exact value of a rounded floating token. Computed named constants, built-in pi/e constants and unsupported powers can prevent proof export. Unsupported models retain native discovery and must report that no certificate was established.

Budgets and retained files

--seconds is the overall search allowance, not a hard real-time deadline. --discovery-seconds allocates native discovery and --ibex-seconds can cap the backend phase. --certify and --ibex-only are mutually exclusive. Certification with --seconds 0 still has finite default phase allocations; see engine/CERTIFICATION.md before changing them.

Each backend call receives unique paths. The Windows backend uses a process watchdog and job isolation. Retained logs, models and coverings may contain confidential equations; set an application retention policy. A report of no_roots_in_certification_domain with completed coverage is a finite-domain result, while no_solution_found without coverage is not a proof that no root exists.

14 Desktop host and appearance controls

desktop/main.cpp hosts wxWidgets and worker lifecycle; desktop/model.hpp handles projects, settings, notation and generic exports. desktop/field_plot.hpp implements field maps, contours and playback; desktop/studies_panel.hpp implements convergence/eigenpair/sweep views. The Desktop launches the neighbouring CLI rather than using eq_solve, preserving the distinct DLL entitlement boundary.

A run captures model/settings/bounds, writes a temporary model and invokes the CLI with --json. The current Desktop run-model limit is 4 MiB; ReadText accepts UTF-8 files up to 5 MiB and project model validation uses 4 MiB. stdout and stderr each have a 64 MiB buffer cap; exceeding it stops the worker. A large sweep can therefore meet engine sample limits yet exceed host response capacity.

Project and result state

.pdeq projects use format patodia.equations.project.v1 with model, settings and bounds. Bound override endpoints and some settings are preserved as text to avoid unnecessary formatting loss. Plain text files save equations only. Project parsing validates into temporary state before assigning the loaded model and settings.

Editing the model does not silently rewrite the last completed result snapshot. During another run, the UI identifies any previous results as belonging to that completed run. Stop terminates the worker process tree; the current protocol does not return a partial JSON solution set after forced interruption. Retained previous results must remain clearly labelled.

Desktop defaults differ from the CLI: seconds=2, starts=30000, solutions=100 and integer neighbors=2 are present in pd::Settings. The desktop passes explicit settings, so changing engine defaults alone will not necessarily change desktop behavior. Avoid introducing a control that the option builder never forwards to the engine.

Fonts and screenshots

Equation and solution fonts are independent. Font family/style and point size controls are saved for the current Windows account rather than inside equation files. The supported size range is 8 to 48 points; reset restores 12 points. Reports and diagnostics follow the solution font, and grid labels, row heights and widths adapt to it.

applyEquationFont changes font properties without clearing syntax colors. applySolutionFont updates the grid and text reports, while resizeSolutionCells handles readable sizing. Font changes must not modify numerical values, project equations or active search settings. The GUI test option checks those preservation properties when a working display is available.

Extending the interface

Keep process execution asynchronous and drain both output streams. Do not execute a concatenated shell command containing model text. If adding settings, update pd::Settings, validation, settings serialization, options, load/save tests and UI bindings together. If adding result columns, use the JSON variable order and retain decimal strings; formatting a root for display must not overwrite the exported value. Chapter 45 covers regression checks for the Equation Generator and Fun with Mathematics.

15 Testing and regression design

A correct maintenance change needs evidence at the same boundary users exercise. Compile the shared source, test CLI behavior, compare DLL results and run desktop model helpers. For syntax or exact

arithmetic changes, use independent mathematical or brute-force oracles instead of asserting only that the implementation returns the same answer as itself.

From the extracted source root with Python 3 available:

```
engine\bin\equation_solver.exe --self-test
engine\bin\equation_solver.exe --self-test --extend
python tests\integer_modular_search.py ^
  engine\bin\equation_solver.exe
python tests\modulo_operator.py engine\bin\equation_solver.exe
python tests\integer_affine_factors.py ^
  engine\bin\equation_solver.exe
```

CMake registers these integer and modulo suites when Python is found. The older engine/run_tests_windows.cmd also invokes bin/windows_dll_smoke.exe. A clean source rebuild does not itself create that client, so build it first if it is absent:

```
gcc -Iengine engine\tests\windows_dll_smoke.c ^
  -Lengine\bin -lequation_solver ^
  -o engine\bin\windows_dll_smoke.exe
engine\run_tests_windows.cmd
```

Use engine/run_ibex_tests_windows.cmd and run_certification_tests_windows.cmd only with an installed, configured backend. Build desktop_model_tests for project/settings/export checks. Enable font_appearance_tests only with wxWidgets and a working interactive display or equivalent test display.

Historical evidence for the affine factor release

Check	Recorded result
New affine factor runs	137, including 80 brute-force comparisons
Existing integer runs	53, including 24 brute-force comparisons
Existing modulo runs	51, including 20 brute-force comparisons
Embedded self-tests	206 double checks and 209 extended checks
Native DLL and CLI parity	15 runs across arithmetic modes
Windows compilation	EXE and DLL cross-compiled; imports and ABI exports checked
Windows runtime and GUI	Not executed; Wine socket creation was blocked

These are historical release results from AFFINE_FACTOR_VALIDATION.json, not a claim that every check was repeated while writing this guide. Cross-compilation cannot substitute for a local Windows launch, loader, licensing and GUI smoke test.

Tests to add with a new recognizer

Exercise both equation directions, declaration/equation permutations, positive and negative targets, nonunit coefficients, repeated factors, zero products, nonintegral recovery, contradictory extra equations and explicit bound overrides. Compare exact root sets in all applicable precision modes. Test unsupported cases to ensure fallback remains available and never reports false completeness.

For numerical changes include multiple known roots, close roots, singular cases, invalid function domains and models with no known feasible point. Record both the accepted residual and root coverage against an independent reference. Avoid pass criteria that depend on a fixed millisecond time on a different machine.

16 Packaging licensing and deployment

Ship a complete runtime set. The current engine depends on libquadmath-0.dll and libwinpthread-1.dll; libquadmath also imports libgcc_s_seh-1.dll. These are bundled beside equation_solver.exe, equation_solver.dll and the desktop host. Older notes saying every compiler runtime was statically linked do not describe this release.

The GCC C++ and compiler options include static-libstdc++ and static-libgcc, but that does not remove every transitive DLL import. Inspect the actual PE import tables and recursively check non-system dependencies. Keep runtime licences/notices with the package.

Licence behavior for integrators

All compute entry points enforce entitlement. The current automatic trial lasts 14 days from current-account first use. A default activated licence is stored under %LOCALAPPDATA%\PatodiaEquationSolver\license.txt on Windows. --license PATH or EQ_LICENSE_FILE selects an explicit file; an invalid explicit selection does not fall back to the automatic trial.

Commercial access supports double-precision CLI/desktop calculation and does not permit DLL solve. Professional, Trial, Academic and OEM have extended/DLL capability subject to their licence terms. Institutional Academic scope and legacy machine-bound licences have different binding rules. DLL capability or academic use does not itself grant redistribution rights.

Licences are checked when computation starts. Keep public verification code and public keys in client builds, but never ship private signing keys or an owner-only issuer kit. Treat metadata and diagnostics as potentially identifying data. Build and release workflows must not delete trial or activation state in an attempt to make an update work.

Release procedure

- Build CLI and DLL from the same reviewed engine source, and keep the public header with the matching ABI.
- Copy the engine into the desktop runtime and confirm the desktop launches that copy.
- Include all runtime dependencies, examples, documentation and required third-party notices.
- Record SHA256 hashes, compiler identity, flags, source snapshot and executed test results.
- Create standard ZIP archives and verify CRCs, extraction and extracted file hashes.
- On Windows, test from a newly extracted folder, including paths with spaces, licence status, CLI solve, DLL load and GUI solve/stop.

The current source archive also includes the ready-to-run dist folder. The command-line runtime package includes the DLL, header and MinGW import library; the desktop runtime adds the GUI and assets. A release can retain an unchanged desktop host while updating its worker, but should state both identities explicitly.

For services, treat a DLL crash as a host-process failure; use process workers where isolation is required. Choose an external process-tree timeout policy for native-first certification, and control retained backend artifacts. Performance measurements should include total wall time, not only time to the first root.

17 Troubleshooting

Symptom	Checks and response
Compressed folder is invalid	Compare download size/hash, use full extraction and verify the ZIP CRCs. Do not reuse a truncated archive.
Windows cannot load EXE or DLL	Check x64 architecture and all transitive runtime dependencies beside the executable.
DLL function pointer crashes	Verify ctypes/PInvoke declarations, cdecl, pointer-sized handles and result lifetime.
DLL status 3 or license_error	Inspect active entitlement and selected licence path; compiled binary128 support is not permission.
GUI still gives an old answer	Confirm the updated equation_solver.exe is beside that GUI and the grid belongs to the current completed model.
Syntax valid but no roots	--check does not search. Run a solve and inspect reason, domains and resource limits.
No solution found for integer products	Confirm the affine update is installed; use integer literals and inspect unsupported-path fallback.
Expected multiple roots are missing	Check hard bounds, merge tolerance, search duration/starts and any returned-point limit.
Modulo answers appear too few	Determine whether rows are residue representatives or a bounded subset; inspect periods and stopping reason.
Extended values lose digits in a host	Keep coordinate/residual strings and use an adequate decimal type rather than double conversion.
Interval result remains incomplete	Check supported syntax, proof domain, unresolved regions, point limits and backend time; do not relabel it complete.
CMake asks for wxWidgets	Configure engine alone if you are building only CLI/DLL.

A useful bug report

Include the smallest model that reproduces the issue, exact option tokens, source/runtime identity, host architecture, licence edition without private credentials, exit code, stdout, stderr and result JSON. For a missing root, provide the expected candidate and independently substitute it into the original equations. For a claimed count, specify the domain and why the reference set is complete.

Separate a loader failure from an engine error and a timed-out run from an empty successful response. Save original model bytes before automatic number-style conversion. When comparing old and new binaries, use matching options and explicit bounds so a changed default does not masquerade as an algorithm regression.

For confidential equations, prepare a smaller synthetic reproducer when possible and review diagnostics before sharing. Never attach a private signing key or activation material merely to report a numerical problem.

18 Source references and extension checklist

The source basis is the corrected Patodia_Equation_Solver_3_0_1_Complete.zip, retaining public C ABI v1. The attached 2.0.1 PDF was checked against its matching editable source before revision. Sections 25 to 35 identify current modules, numerical contracts, public routing and release requirements. Record actual source/binary hashes rather than treating a filename as an immutable identity.

Contract or behavior	Primary files
Public functions and ownership	engine/equation_solver.h; engine/src/library.cpp
CLI parsing and stdout	engine/src/main.cpp; engine/src/equations.tpp
Models, domains and ordering	engine/src/equations.hpp; engine/src/equations.tpp
Precision and numerical search	engine/src/common.hpp; equations.hpp; equations.tpp
Integer polynomial search	engine/src/integer_search.hpp
Affine product recognition	engine/src/integer_factors.hpp; tests/integer_affine_factors.py
Mathematical remainder	engine/src/modulo_search.hpp; MODULO_OPERATOR.md
Proof orchestration	engine/src/interval_proof.hpp; engine/CERTIFICATION.md
Backend process adapter	engine/src/ibex_backend.hpp; engine/IBEX_INTEGRATION.md
Licence and trial checks	engine/src/license.hpp; automatic_trial.hpp; engine/LICENSE_AND_STYLE.md
Desktop projects and exports	desktop/model.hpp; tests/model_tests.cpp
Font controls	desktop/main.cpp; FONT_CONTROLS.txt; tests/font_appearance_tests.cpp
Build and package facts	Build CMD files, CMakeLists.txt and AFFINE_FACTOR_VALIDATION.json
Numerical Version 3 models	field_v3.hpp; quadrature.hpp; equations.tpp dispatcher
Desktop numerical views	desktop/field_plot.hpp; desktop/studies_panel.hpp

Before changing a contract

Identify whether the change affects model syntax, numeric behavior, ABI, JSON schema or desktop project format. An internal optimization can preserve public interfaces; an added exported function, changed pointer contract or altered field type requires explicit compatibility review. Do not change a JSON string coordinate into a number to simplify one language binding.

When adding a function or operator, review expression parsing, evaluation, derivatives where applicable, integer/modulo eligibility, proof export and error diagnostics together. Unsupported paths should decline explicitly and preserve the existing safe fallback. Add a regression at the public entry point rather than testing only a private helper.

When adding an option, keep CLI token handling, DLL allowlist, validation, help text and desktop option forwarding aligned. If the option takes a value, test missing and invalid values and interactions with precision and certification. An option accepted by the CLI is not automatically accepted by the DLL.

Before release, repeat root-set tests, original-residual checks and CLI/DLL parity. Preserve licensing, cancellation/budget behavior, decimal serialization and result ownership. Update the developer guide and the validation record to say what was tested, on which platform, and what remains unverified. These details allow the next developer to reproduce both the behavior and its limits.

19 Exact quadratic and large integer contracts

The general two-variable integer route recognises one quadratic identity A*x^2+B*x*y+C*y^2+D*x+E*y+F=0 with supported exact integer coefficients. It complements the affine-product shortcut and the positive Pell and weighted-square routes. Use int for signed and zero branches, or pos int for strictly positive coordinates. Additional equations, inequalities and unsupported coefficient expressions can prevent this specialised route from applying.

```
pos int x,y
x^2-2*y^2=1
bound x=[1e499,1e500]
```

In ordinary double mode this model bounds x inclusively between 10^499 and 10^500. The recognised route compares simple decimal or scientific bound literals exactly, without passing their magnitude through double. Full decimal endpoints also work. Lower bounds are rounded upward and upper bounds downward for integer domains. Type restrictions still intersect the effective bounds; CLI named overrides take precedence over global and model bounds.

Pell equations and weighted squares

```
int x,y
x^2-61*y^2=1
bound y=[-1000000,1000000]
```

Generalised Pell input x^2-D*y^2=N accepts supported integer D and N through the exact quadratic machinery. Do not infer a complete unbounded family from a finite returned list. Positive-only Pell routes and general signed quadratic routes can use different completion fields. Read the selected search_arithmetic, reason and applicable integer completion flag.

```
pos int x,y
2*x^2+3*y^2=35
```

This weighted-square example has (x,y)=(2,3) and (4,1). The positive weighted-square algorithm uses exact preprocessing, certified factorisation, modular square roots and Cornacchia-style descent, including nonprimitive representations. The signed general-quadratic route handles broader conics using exact reductions, congruence recovery and bounded slicing where needed. Difficult factorisation, degenerate unbounded families and resource caps remain limitations.

Exact values and completion

precision=exact_integer and exact_integer_solutions=true describe verified integer identities. integer_search_complete or positive_integer_search_complete describes enumeration in the applicable integer domain. search_complete retains its separate real interval-certification meaning. A solution cap, time stop or incomplete factorisation must not become a proof of absence. A bound limits magnitude; --max-solutions limits returned rows. Use a sufficiently large row cap when requesting every point in a bounded interval.

Oversized literals are supported only by recognised exact two-variable quadratic, positive Pell and positive weighted-square paths. Unsupported real, mixed-integer, modulo or higher-dimensional numerical models reject them. Expressions such as 10^500 or (1e500) are not a new arbitrary-precision

bound expression language. Literal guards include 10000 characters, 10000 magnitude digits and exponents in the range plus or minus 10000. Equation-coefficient parser and recogniser limits remain separate.

Long value display and preservation

Desktop 2.0.1 retains full coordinate strings in grid data and exports. Columns containing more than 120 characters are capped at 520 device-independent pixels; ordinary cells retain their established sizing. A double-click opens the intended full-value viewer. The user has reported that the viewer shows only part of the solution on Windows. No corrective viewer release is established by this guide; treat this as an open display defect. Preserve existing Ctrl+C and inspect CSV or JSON strings when checking whether all digits survive. Portable string tests and GUI compilation do not establish rendered accessibility.

20 Initial value ODE integration

```
ode
y''+0.5*y'+4*y=0
initial t=0
initial y=1
initial y'=0
interval t=[0,20]
method=RK45
output step=0.05
rtol=1e-8
atol=1e-11
```

The independent variable is named by interval. Supply the initial independent value at the first endpoint and initial values through derivative order m−1 for each order-m equation. Direct orders 1 through 8 expand to at most 64 first-order states. Equations must be affine in their own highest derivative; isolate coupled highest derivatives before input. This is not a DAE or arbitrary implicit-branch solver. Backward IVP integration is supported.

Method	Computation	Accuracy contract
RK4	Fixed classical fourth order; positive step required	No adaptive error estimate; compare smaller steps
RK45	Dormand–Prince 5(4) with quartic dense output	rtol and component atol control estimated local error
SDIRK2	L-stable second order implicit stages and step doubling	Newton iteration and interpolation checks; no automatic stiffness switching

Defaults are rtol=1e-7, atol=1e-10 and max steps=100000. SDIRK2 defaults to 12 Newton iterations. Output step and output points control reporting, not adaptive internal steps. Choose one reporting specification: output step, output points or ordered sample times. RK4 output step must be an integer multiple of its fixed step. Output has a 9999 requested-row limit and a 500000 display-cell limit.

Events and continuous output

```
event ground=y
event ground terminal=true
event ground direction=falling
event tolerance=1e-10
save dense=true
```

Event brackets use sign changes on accepted continuous segments. Tangencies or multiple crossings inside one segment may be missed; denser reporting alone does not fix this. Directions refer to increasing independent-variable time even for backward integration. Terminal stopping has

integration_successful=true and status event_terminated, but integration_complete is true only if the requested endpoint is reached.

Saved dense_segments contain t0, t1, start, end and coefficient rows c. For u=(q−t0)/(t1−t0), evaluate start+u*(c[0]+u*(c[1]+...)); use stored endpoint values at exact endpoints. Never extrapolate. Do not assume a fixed number of coefficient rows: singular-series output can contain more than the RK polynomial degrees. Saving is optional and capped at 10000 segments and 500000 coefficient cells.

IVP results use problem_type=ode_ivp and equation_solver.ode_result.v1. Read trajectory rather than solutions. Partial numerical failures can retain earlier finite samples; inspect integration_successful, integration_complete, status and reason. An initial row alone is not integration success. --extend runs ODE integration in binary128, but does not automatically tighten the ODE tolerance defaults. Desktop plots still use double precision.

21 Automatic singular IVP continuation

```
ode
x*y''+y'+x*y=0
initial x=0
initial y=1
initial y'=0
interval x=[0,1]
step=0.1
```

This original Bessel input retains x=0, y=1 and y'=0 and returns y(1) approximately 0.7651976865579666. The regular expansion is y=1−x^2/4+x^4/64−… and y''(0)=−1/2. The engine uses regular_singular_series across the supported interval; it does not merely move the start to an arbitrary epsilon or claim to execute classical RK4 stages there.

The supported class is scalar linear first- or second-order IVPs with simple affine leading coefficient P(x)=a*(x−s), nonzero a, and polynomial remaining coefficients of degree at most eight after supported exact rational manipulation. Some explicit affine-denominator forms and removable interior holes are recognised. Initial, final and interior singular points can be analysed in either integration direction. Interior continuation must fit the original initial data and required classical smoothness.

Outcome	Diagnostic
Compatible determined regular branch	regularization_applied=true; method=regular_singular_series
Incompatible finite regular data	singular_incompatible
Free continuation coefficient or classical branch freedom	singular_ambiguous
Unsupported structure or branch analysis	singular_unsupported
Insufficient series truncation or roundoff budget	singular_series_limit

For x*y'=y with y(0)=0 the family y=C*x is ambiguous. Genuine divergence cannot be repaired by replacing the singular coordinate. Singular BVPs, coupled singular systems, nonlinear singularities, logarithmic/fractional branches and arbitrary symbolic limits remain outside this feature.

The recurrence is constructed through degree 64 and selects a degree 16, 32, 48 or 64 representation using tail and roundoff estimates over the interval. Component atol governs these absolute checks; they are not certified global error bounds. JSON preserves requested_method separately and includes regularization diagnostics, dense_interpolant and adaptive_error_control=false for this route. A syntax check alone does not establish compatibility.

22 Two point boundary value problems

```
bvp
y''-2*x*y'-2*y=-4*x
interval x=[0,1]
boundary left y-y'=0
boundary right 2*y-y'=1
output points=101
```

This Robin BVP has the exact solution y=x+exp(x^2). A second requested example is shown below; with L=ln(1.4), its analytic solution is y=(ln(x))^2/2+(1−L)*ln(x)/2. Both are regular on their specified intervals.

```
bvp
x^2*y''+x*y'=1
interval x=[1,1.4]
boundary left y=0
boundary right y=ln(1.4)/2
output points=101
```

The method is fourth-order Lobatto collocation with damped Newton solves and adaptive mesh refinement. Supported finite intervals require a<b. Orders up to eight expand to at most 16 first-order states; exactly that many independent scalar boundary conditions are required. Endpoint shorthand evaluates its whole expression at that endpoint. Coupled conditions use left(expression) and right(expression), for example boundary left(y)−right(y)=0.

Optional guess y=... and derivative-component guesses seed a local solve across the interval; missing guesses default to zero. Different guesses can converge to different nonlinear branches. This is not enumeration or a uniqueness proof. Initial guesses are independent component expressions; derivative guesses are not automatically differentiated from y.

Defaults are rtol=1e-6, atol=1e-8, boundary tolerance=1e-8, mesh points=11, max nodes=1001, newton iterations=25 and max refinements=16. Hard limits include 4000 nodes, 6000 nodal unknowns, 500000 sparse entries and 500000 output cells. Reporting density is separate from mesh refinement. Defect and boundary residual acceptance are numerical criteria rather than global error guarantees.

BVP results use equation_solver.bvp_result.v1, problem_type=ode_bvp and method=lobatto_collocation_4. Read bvp_successful, boundary_residuals and maximum_scaled_defect. Convergence returns status bvp_converged and code 0. Numerical failure returns code 2 and an empty trajectory; no unconverged curve is exposed as a solution. Singular endpoints, unknown eigenparameters, multipoint conditions and DAEs are outside this regular BVP mode. Singular IVP support does not extend that scope.

23 PDE and integral equation integration

Rectangular Poisson and Laplace equations

```
pde
type=poisson
u_xx+u_yy=0
domain x=[0,1]
domain y=[0,1]
boundary left u=sin(pi*y/2)
boundary right u=0
boundary bottom u=0
boundary top u=1-x
grid x=17
grid y=17
```

The model illustrates compatible corner values, not an asserted closed form. The unknown is u and coordinates x,y. The supported steady operator is a*(u_xx+u_yy)=f with equal constant nonzero diffusion coefficients. The current method is conservative rectangular finite volumes with natural normal/Robin contributions and sparse preconditioned CG. Dirichlet corner values must agree; all-normal data additionally require a mean and compatibility checks.

Heat and wave equations

```
pde
type=heat
u_t=u_xx
domain x=[0,1]
interval t=[0,0.1]
boundary left u=0
boundary right u=0
initial u=sin(pi*x)
grid x=41
grid t=101
```

The heat reference is exp(-pi^2*t)*sin(pi*x). Linear coefficients can depend on x,t with strictly positive diffusion. Crank-Nicolson uses centred diffusion and tridiagonal solves; heat retains centred advection only under its cell-Peclet check. Select the explicit upwind advection_diffusion class for that scheme. Implicit stability does not guarantee positivity or resolution.

```
pde
type=wave
u_tt=u_xx
domain x=[0,1]
interval t=[0,1]
boundary left u=0
boundary right u=0
initial u=sin(pi*x)
initial u_t=0
grid x=41
grid t=101
```

The wave example has u=cos(pi*t)*sin(pi*x). The supported form is u_tt=c^2*u_xx+f(x,t), constant positive c^2. An explicit centred scheme requires c*dt/dx<=1 and returns unstable_grid if that check fails. Initial position, velocity and boundary corner compatibility are checked. No variable-speed, damped or nonlinear wave solver is implied.

Second kind integral equations

```
integral
type=fredholm
interval x=[0,1]
forcing=1+x/6
kernel=x*s
lambda=1
grid x=65
```

The Fredholm example has y=1+x and uses composite-trapezoidal Nystrom discretisation with a dense pivoted solve. Volterra with forcing=kernel=1 has y=exp(x) and forward trapezoidal marching. Automatic refinement now returns Richardson-extrapolated values rather than the raw requested-grid solve. Coupled linear Volterra systems support up to eight components; Fredholm remains scalar.

Integral equations require finite intervals and ordinary finite smooth kernels. First-kind inversion, nonlinear/singular/weakly singular kernels, principal values and unknown parameter fitting are excluded. This restriction is separate from quadrature, which supports known-function singular

endpoints and infinite limits. A singular discrete pivot is a diagnostic, not a universal continuous existence theorem.

Refinement and result interpretation

refine=true now repeats finer internal solves until an estimated absolute-plus-relative target is met or limits intervene. PDE fine-grid differences use divisor 3 for second-order schemes and 1 for first-order upwind spatial convergence. Integral successive Richardson extrapolants use divisor 15. Requested output coordinates are retained, with finest computed values sampled there. refine=false sets accuracy_target_met=null; accuracy_not_met can retain samples with field_successful=false.

Default absolute/relative targets are each 1e-3 for PDE/eigen and 1e-9 for integral equations. max grid defaults: PDE 801, eigen 1025, Fredholm 513, Volterra 4097. max nodes defaults to 250000 internal nodes, allowed up to 500000; output is capped at 10000 nodes. Requested PDE spatial counts are 3..401 (time-PDE x at least 5), time counts 2..10001, integral x 3..513 and eigen x 3..1025. Linear tolerance defaults to 1e-11 double or 1e-25 binary128; field boundary tolerance to 1e-8.

Field schema v1 now covers problem_type=pde, integral_equation or eigenproblem. Read field_successful, accuracy_target_met, status, reason, grid_shape, internal_grid_shape, trajectory, convergence and optional eigenpairs/exact-error fields. solutions stays empty. PDE row order is x fastest, then y or time. Error estimates concern returned nodes under stated smooth convergence assumptions; they are not certified global bounds. Arbitrary geometry, periodic/nonlinear/coupled PDEs and multidimensional transients remain excluded.

24 Result routing and licensing acceptance

Result family	Schema	Data and success checks
Algebraic and exact integer	equation_solver.result.v1	solutions; applicable exact integer completion flags
ODE IVP	equation_solver.ode_result.v1	trajectory; integration_successful and integration_complete
ODE BVP	equation_solver.bvp_result.v1	trajectory; bvp_successful
PDE or integral	equation_solver.field_result.v1	trajectory and convergence; field_successful and accuracy_target_met; fixed-grid targets are null
Eigenproblem	equation_solver.field_result.v1	eigenpairs and trajectory; field_successful; eigenvalue accuracy target
Parameter sweep	equation_solver.sweep_result.v1	runs[].result; all_runs_successful and per-run status
Quadrature	equation_solver.quadrature_result.v1	value; quadrature_successful and accuracy_target_met; candidate is unaccepted

Keep numeric strings and JSON nulls intact. Empty solutions is normal for numerical families. Read the process/DLL code, schema, problem_type and mode-specific acceptance fields. A fixed-grid field completion has an unassessed target; a quadrature candidate is unaccepted when value is null. C ABI v1 still has ten exports and unchanged borrowed-string ownership.

Commercial command line use is permitted

A Commercial licence permits both Desktop and command-line double-precision solving. It does not permit extended precision or DLL solving. The Desktop itself uses the CLI worker, so allowing the CLI is consistent with the current design. Professional, Trial, Academic and OEM support extended precision and DLL computation subject to their contracts; technical entitlement does not grant redistribution rights. Loading the DLL, querying its ABI or checking metadata is not a computation entitlement test.

```
equation_solver.exe --license-status
equation_solver.exe --eq "x^2=2" --json
equation_solver.exe --eq "x^2=2" --extend --json
```

Under a valid Commercial licence, the standard solve should be allowed and the extended solve should fail with an entitlement error. Also call eq_solve through the DLL and assert rejection; do not merely test whether Windows loads it. An activated licence with expires_unix=0 is perpetual. automatic_trial=false indicates the reported entitlement is not the automatic trial.

Default activated licence path: C:\Users\<WindowsUserName>\AppData\Local\PatodiaEquationSolver\license.txt. Use %LOCALAPPDATA%\PatodiaEquationSolver to open the folder. --license PATH and EQ_LICENSE_FILE override selection. An explicitly selected missing, invalid, expired or wrong-machine individual licence must be rejected without trial fallback. With no selected licence file, an eligible current account automatically receives a 14-day trial; absence of license.txt alone does not establish a bypass.

Windows trial state is stored as protected trial.dat in that directory and TrialStart under HKEY_CURRENT_USER\Software\PatodiaInfotech\EquationSolver. Clock state is also protected. Preserve these states during updates. Test tampering, expiry and clock behaviour in controlled QA environments. Never include issuer private keys in a runtime or public source distribution.

25 Current source map and release validation

Current source files	Responsibility
integer_bound_literals.hpp	Exact decimal literal bounds shared with Desktop
pell_search.hpp and general_pell.hpp	Pell recognition and exact recurrences
quadratic_search.hpp	General signed two-variable integer quadratics
sum_squares_search.hpp and sum_squares_arithmetic.hpp	Weighted squares and factorisation or modular machinery
ode.hpp and ode_prepare.hpp	IVP syntax and higher-order conversion
ode_integrate.hpp and ode_numerics.hpp	Integration, dense output, events and implicit numerics
ode_singular.hpp	Supported regular singular recognition and series
bvp.hpp and bvp_collocation.hpp	BVP preparation and adaptive collocation
field_equations.hpp	PDE and integral parsing, discretisation and results
desktop/main.cpp and desktop/model.hpp	Worker UI, result routing, plotting, exports and exact bounds
field_v3.hpp	Current field/eigen/integral/sweep algorithms and result assembly
quadrature.hpp	Standalone quadrature, error/tail checks, scaled evaluation
desktop/field_plot.hpp; desktop/studies_panel.hpp	Numerical display and exports

Engine paths in the table are relative to engine/src unless prefixed by desktop. The vendored header-only Boost subset is under engine/third_party; retain its copyright and licence files. CLI, DLL and Desktop need

no Python, SciPy or external PDE runtime for the numerical modes described here. Development reference tests can use Python and SciPy.

Recorded cumulative validation

The corrected numerical release audit records 69 quadrature cases per precision mode through CLI and DLL, 43 Version 3 cases per mode/interface, 104 field CLI cases, 16 field DLL cases, 15 scalar-root checks and 23 teaching examples. Earlier 2.0.1 and affine-factor records remain historical evidence, not totals for the new build. These counts are release records; this guide update does not claim to rerun every suite.

The corrected Windows engine EXE and DLL were cross-compiled and PE/version/import/export inspected. The compatible Desktop binary was retained unchanged. Windows execution and GUI/account-specific licence acceptance remain separate tasks. A native Linux pass is not a Windows runtime pass, as the open endpoint-cancellation case illustrates.

Acceptance cases before the next release

Run exact Pell bounds around 500- and 1000-digit endpoints, compare against an independent recurrence and check row-cap completion diagnostics. Verify copied and exported values byte for byte. On Windows, resize the full-value viewer at several fonts and display scales, scroll to the final digit and verify both selection copy and Copy full value. Keep ordinary column sizing and existing Ctrl+C behaviour unchanged.

Test Bessel startup and singular continuation, BVP branches/failures, mixed-boundary PDEs, Neumann compatibility, wave CFL checks, eigenvalue/mode normalisation, integral extrapolation, coupled Volterra components, parameter-sweep limits and quadrature endpoint/infinite-tail diagnostics. Include fixed-grid success, accuracy_not_met with samples, null quadrature value with candidate, malformed input and cancelled/time-limited work.

Test Commercial CLI success, Commercial extended and DLL denial, eligible automatic trial, expired trial and invalid explicit licence rejection using the actual packaged Windows binaries. Preserve the licence policy when changing the viewer. Record source identity, runtime hashes, compiler flags, actual tests and platform limitations with the release.

Authoritative support notes

Use VERSION3_GUIDE.md, QUADRATURE_GUIDE.md and QUADRATURE_OVERFLOW_FIX.md with the actual source, release audit and validation logs. Earlier exact-integer, IVP/BVP and singular-IVP notes retain applicable detail. The authoritative declarations are engine/equation_solver.h; option allowlisting is engine/src/library.cpp. Update this guide whenever model syntax, JSON semantics, resource guards, export formatting or build/version handling changes.

26 Boundary aware field algorithms and accuracy contracts

field_v3.hpp builds on the shared FieldInput, expression and numerical primitives in field_equations.hpp. The numerical suite is included inside Engine<Real>; do not move these headers to a standalone translation unit without resolving their enclosing-template dependencies.

Boundary representation and signs

v3Boundary converts exactly one u, normal or robin directive per side into V3BC alpha, beta and g expressions. V3BC::at checks nonnegative alpha/beta and excludes both zero. Robin means

alpha*u+beta*du/dn=g. Normal derivatives are -u_x at left, +u_x at right, -u_y at bottom and +u_y at top. Physical conductivity is not silently included in a normal directive.

```
pde
type=poisson
u_xx+u_yy=4
domain x=[0,1]
domain y=[0,1]
boundary left normal=0
boundary right robin=[1,1,3+y^2]
boundary bottom u=x^2
boundary top u=x^2+1
grid x=9
grid y=9
exact solution=x^2+y^2
```

For u=x^2+y^2 the source is 4, the left normal is zero and the right Robin value u+du/dn is 3+y^2. This checks value/derivative/Robin assembly without assuming that a small algebraic residual is a continuum certificate.

Pure Neumann compatibility and mean

All-normal Poisson/Laplace data need mean value to remove the additive nullspace. Separate composite-Simpson integration assesses continuous source/normal-flux compatibility. incompatible_conditions and compatibility_not_resolved have different meanings. The discrete trapezoidal finite-volume right-hand side is projected to zero balance; neumann_compatibility_correction reports the smooth O(h^2) uniform source adjustment. A mean does not cure incompatible flux data.

Refinement state and returned values

v3Single computes an initial solution, doubles internal subdivisions and samples each finer solution at the original requested coordinates. For PDEs the estimated fine error is the sampled difference divided by 3 (second order) or 1 (upwind first-order spatial convergence). Integral equations remove the leading trapezoidal term and compare successive extrapolants with divisor 15. Eigenvalue targets apply to the reported eigenvalues, not eigenfunction derivatives.

```
absolute tolerance=1e-6
relative tolerance=1e-6
max refinements=4
max grid=801
max nodes=250000
refine=true
```

Each assessed component must meet absolute tolerance+relative tolerance*abs(value). Defaults are 1e-3 for each PDE/eigen target and 1e-9 for each integral target. max refinements defaults to 4 for PDE/eigen and 6 for integral, allowed 1..10. Integral processing uses one preliminary refinement before its first estimate-producing extrapolant comparison.

Field	Contract
grid_shape	Requested returned coordinate grid
internal_grid_shape / internal_grid_points	Actual internal grid; internal_grid_points is the x count
convergence[]	level, grid_shape, estimated_error, observed_order, discrete_residual and optional exact_max_error
residual_scope	internal_discrete_solve
error_estimate_scope	returned_nodes_smooth_asymptotic_assumption
refine=false	Fixed-grid success; accuracy_target_met=null

Field	Contract
accuracy_not_met	field_successful=false; computed samples may remain

Retain JSON null rather than converting it to false or zero. observed_order may be null or unreliable before asymptotic convergence or near roundoff. Exact reference fields are optional diagnostics: exact_max_error and exact_rms_error use returned samples. They are not required to solve general models without closed forms.

27 Regular Sturm Liouville implementation

The eigen mode solves -(p*y')'+q*y=lambda*weight*y on a finite regular interval. Positive p and weight are required; separated homogeneous Dirichlet, normal and supported nonnegative Robin conditions are accepted. Nonhomogeneous, periodic, singular-endpoint and non-self-adjoint eigenproblems are outside this implementation.

```
eigen
type=sturm_liouville
interval x=[0,1]
p=1
q=0
weight=1
boundary left u=0
boundary right u=0
modes=3
grid x=33
exact eigenvalue=(n*pi)^2
exact eigenfunction=sin(n*pi*x)
```

The symmetric flux stiffness and lumped weight matrix are transformed to a symmetric tridiagonal problem. Sturm counts isolate the lowest requested eigenvalues; shifted inverse iteration with orthogonalisation constructs modes. Discrete eigenpair residual checks guard acceptance. Requested modes default to 4 with range 1..20; output x defaults to 65 and internal max grid to 1025.

Normalisation and reference fields

Internal trapezoidal weighted-L2 normalisation makes the weighted discrete sum of y^2 equal one, and the first significant mode sample is made positive. Names mode1, mode2 and so on identify returned mode columns. Each eigenpairs record includes index, eigenvalue, discrete_residual and normalization; optional references add exact_eigenvalue, absolute_error and exact_function_max_error.

exact eigenvalue expressions use one-based n; exact eigenfunction expressions use x,n. The reference eigenfunction is normalised by the same internal quadrature and sign-aligned before comparison. Neumann-Neumann problems with p=weight=1 and q=0 begin with a zero constant mode, so their reference formulas use n-1. The acceptance target concerns eigenvalues, not a certified eigenfunction-error bound.

A weighted norm or an arbitrary eigenfunction sign is not a physical amplitude. Hosts should retain mode indexing and normalisation metadata when comparing runs, and avoid interpreting lambda as frequency without the original physical scaling. Unit conversion and material-property interpretation remain host/model responsibilities.

28 Linear transport and coupled Volterra systems

Version 3 adds explicitly named reaction_diffusion and advection_diffusion types and extends Volterra to coupled linear components. These modes share field parsing and refinement but have distinct numerical order and coupling contracts.

Reaction and advection diffusion

```
pde
type=reaction_diffusion
u_t=u_xx/10-2*u
domain x=[0,1]
interval t=[0,0.1]
initial u=sin(pi*x)
boundary left u=0
boundary right u=0
grid x=17
grid t=41
exact solution=exp(-(pi^2/10+2)*t)*sin(pi*x)
```

Reaction-diffusion uses centred diffusion and Crank-Nicolson time stepping with second-order smooth convergence. The state equation must remain linear; the leading time coefficient is a nonzero constant and diffusion is strictly positive. Coordinate-dependent linear coefficients and sources are accepted. Nonlinear reaction, coupled PDE species and arbitrary geometry are not supplied by this class.

advection_diffusion adds first-order upwind spatial advection with Crank-Nicolson time stepping; its refinement divisor is 1. The heat class retains centred advection only when its cell-Peclet check is at most 2. Coarse Crank-Nicolson stepping can fail to preserve positivity even when a linear stability criterion holds. Do not market implicit stepping as an accuracy or positivity certificate.

Coupled Volterra input

```
integral
type=volterra
components=2
interval x=[0,1]
forcing 1=1
forcing 2=2
kernel 1 1=0.5
kernel 1 2=0.25
kernel 2 1=1
kernel 2 2=0.5
lambda=1
grid x=17
exact 1=exp(x)
exact 2=2*exp(x)
```

The convention is y_i=f_i+lambda*sum_j integral_a^x K_ij(x,s)*y_j(s) ds. All numbered forcing and kernel entries are mandatory, including zero kernels. Up to eight components are allowed; coupled Fredholm is rejected. A pivoted dense endpoint matrix solve replaces independent scalar updates at each marching node. Adaptive extrapolation checks every returned component.

Scalar inputs use forcing, kernel and optional exact solution; systems use forcing i, kernel i j and optional complete exact i references. Mixing scalar and system directives is rejected. A singular diagonal or coupled endpoint matrix reports singular_system; do not hide it by replacing a zero divisor with an epsilon.

29 Parameter sweep execution and JSON batches

runField extracts sweep directives, validates each parameter as a declared constant, builds a Cartesian product and calls v3Single for each instance. This is forward repeated solving, not continuation, optimisation, regression or an algebraic branch tracker.

```
eigen
type=sturm_liouville
const stiffness=1
p=stiffness
q=0
weight=1
interval x=[0,1]
boundary left u=0
boundary right u=0
modes=3
grid x=33
exact eigenvalue=stiffness*(n*pi)^2
sweep stiffness=[1,2,4]
```

Combinations are capped at 64 and total stored trajectory rows at 200000. Dependent constants are recalculated per instance. The complete batch shares one time budget. runs[] contains parameters and a nested result. Inspect all_runs_successful, top-level field_successful and every nested status; sweep_incomplete, time_limit and resource_limit are not successful batches.

Failed attempted cases can contain invalid_model and a reason. A budget or storage stop can leave fewer stored runs than the nominal Cartesian product. Preserve the original model and planned case count so missing/unstarted cases can be distinguished from failed stored cases. Never fill absent cases with zeros or silently discard failures from a comparison table.

Batch representation

```
{
  "schema": "equation_solver.sweep_result.v1",
  "problem_type": "parameter_sweep",
  "sweep_parameters": ["stiffness"],
  "all_runs_successful": true,
  "runs": [
    {"parameters": {"stiffness": "1"}, "result": {}}
  ]
}
```

This is a structural illustration, not a complete response. Each result object contains its own full field/eigen/integral schema. Nested parameters and numerical samples are strings. Sweep support applies to the supported PDE, integral-equation and eigen modes; quadrature explicitly rejects sweep directives.

30 Standalone quadrature parsing and expression evaluation

quadrature.hpp implements known-function one-dimensional definite integration. It does not add integrate(...) to the expression language or solve unknown-dependent integral expressions. Every parameter must have an assigned numerical value.

```
quadrature
integrand=1/(1+x^2)^5
variable=x
interval=[0,inf]
absolute tolerance=1e-10
relative tolerance=1e-10
```

QuadInput validation

QuadInput accepts integrand or expr, variable or var, interval or range, the paired from and to directives, and the optional singularities list for known split points. Accuracy and resource keys include absolute tolerance, relative tolerance, max refinements, max evaluations, max extent and subdivisions. const/constant declarations assign parameters. The variable defaults to x and must be an unassigned valid identifier. The integrand is capped at 4096 expression nodes. Nonliteral zero-valued constants are screened because computed zero can reflect underflow or cancellation.

Endpoints accept case-insensitive inf/infinity tokens with signs, or finite scalar expressions. Reverse limits are swapped internally and negate the result. Equal finite limits give zero width; identical infinite limits are invalid. Infinity tokens are endpoint syntax, not arbitrary expression variables. EU notation is normalised before mode parsing.

Control	Default	Permitted value or role
absolute tolerance	1e-10	Nonnegative; at least one tolerance positive
relative tolerance	1e-10	Nonnegative; target is atol+rtol*abs(I)
max refinements	10	1..16
max evaluations	1000000	32..10000000
subdivisions	4	1..64, finite intervals only
max extent	8	2..12, transformed coordinate range

Ordinary evaluation and scaled fallback

QuadWorker first evaluates the original AST normally. A nonfinite result triggers quadScaledExpression, representing supported intermediate products, quotients, powers and functions with sign and logarithmic magnitude. QuadScaled retains an ordinary raw value when representable; function arguments such as trigonometric inputs therefore retain their ordinary argument reduction where possible. The transform weight is combined before materialising a weighted sample.

This is the build 3.0.1.1 correction for the reciprocal fifth-power improper integral. Its independent reference is 35*pi/256. It is not universal arbitrary-precision evaluation: invalid domains, division by zero, unrepresentable function arguments and unresolved cancellation still fail. loss_of_significance describes cancellation below the logarithmic fallback resolution rather than accepting a guessed zero.

scaled_evaluations counts fallback uses. A scaled-evaluation roundoff allowance enters the error floor. Ordinary finite evaluations keep the established arithmetic path. Do not turn every nonfinite node into zero: doing so can accept a divergent or undefined integral. Add adversarial domain/cancellation tests whenever extending an opcode in this evaluator.

31 Quadrature transforms tail checks and acceptance

QuadLeg identifies finite, right-infinite and left-infinite pieces. Finite intervals are initially subdivided; whole-line integrals become two independent one-sided legs split at zero. No principal-value cancellation is used. Multiple explicit ranges or a declared singularity list can supply separate pieces. See Chapter 45 for inputs and expected results.

Transformed sampling

```
Finite: x = (a+b)/2 + (b-a)/2*tanh((pi/2)*sinh(u))
Right:  x = a + exp((pi/2)*sinh(u))
Left:   x = b - exp((pi/2)*sinh(u))
```

Implementation uses stable finite-endpoint distances and logarithmic weights rather than blindly evaluating the displayed identities. max extent bounds u approximately in [-E,E]; it is not a physical cutoff at x=E. On [0,inf], E=8 can reach x of order 10^1017. Larger extent explores farther tails; refinement changes grid density. Neither alone guarantees that an unsampled feature is resolved.

Independent phases and error floors

QuadWorker caches transformed samples by leg and u. Ordinary and half-shifted phases are compared, together with successive levels, absolute mass, terminal tail groups and roundoff. Acceptance requires consecutive qualifying levels. The error estimate includes phase difference, inter-level change, tail estimate and guard-arithmetic roundoff. Budget checks cover time, cancellation and the maximum evaluation count.

Endpoint clipping, unresolved decay, overflow, underflow and all-zero samples cannot be silently interpreted as convergence. quadExactZero proves only limited structural zeros; other vanishing samples produce unresolved_zero when acceptance cannot establish a nontrivial integral. quadPoleCheck isolates eligible polynomial denominator zeros in the interior, including denominators under selected roots/absolute values. It is not an exhaustive transcendental singularity detector.

Status	Host interpretation
integration_complete	Accepted value with success/target flags
nonfinite_integrand / nonfinite_transform	Expression or weighted evaluation failure
interior_singularity	Detected interior denominator zero; analyse/split the problem
tail_not_resolved	Absolute tail decay not established
unresolved_zero	Samples vanished without sufficient structural proof
precision_limit / roundoff_limit	Arithmetic/endpoint resolution blocked acceptance
refinement_limit / evaluation_limit	Numerical budget exhausted
time_limit / cancelled	Interrupted numerical work

Precision and candidate semantics

runQuadrature delegates standard mode to Engine<__float128>::quadratureResult and labels precision=binary128_guard. Extended mode labels binary128; requested_precision preserves the user selection. The normal extended/DLL licence checks still apply. Printed digits do not imply error certification or hundreds of accurate digits.

On failure value is null. candidate_value, when present, remains explicitly unaccepted. Host code must preserve that distinction even if the candidate matches an independent reference. JSON also reports estimated_absolute_error, estimated_tail_error, roundoff_estimate, convergence, evaluations, endpoint_precision_reached and tail_decay_resolved when available. Failure before an estimate may leave fields absent or null. The 3.0.1 update adds targeted fallback strategies for difficult quadrature cases. Apply the same status and acceptance checks to the final result when a fallback is used.

Open endpoint regression

```
quadrature
integrand=x/sqrt(1-sqrt(x))
variable=x
interval=[0,1]
```

The mathematical reference is 64/35. A Windows run reported division by zero near the endpoint after an accurate candidate; a native Linux rebuild accepted the original model. Rationalising to x*sqrt((1+sqrt(x))/(1-x)) and the smooth substitution integral_0^1 4*(1-s^2)^3 ds supply independent checks. Do not mark the Windows defect fixed from a Linux pass or suppress the undefined evaluation. Stable endpoint evaluation needs an explicit verified correction and regression.

32 Mode aware result routing for host applications

The following Python function classifies completed responses while preserving accepted/diagnostic distinctions. It uses no float conversion and never treats a candidate or a successful syntax check as an accepted numerical answer. Keep the original JSON and external process/DLL code beside the classification.

```
def classify(j, code=None):
    status = j.get("status")
    schema = j.get("schema")
    kind = j.get("problem_type")
    base = {"schema": schema, "problem_type": kind,
            "status": status, "reason": j.get("reason"),
            "accepted": False}
    if status == "syntax_valid":
        return {**base, "kind": "syntax_check"}
    ok_code = code is None or code == 0
    if schema == "equation_solver.quadrature_result.v1":
        ok = (ok_code and j.get("quadrature_successful") is True
              and j.get("accuracy_target_met") is True
              and isinstance(j.get("value"), str))
        return {**base, "accepted": ok, "kind": "quadrature",
                "value": j.get("value") if ok else None,
                "candidate": j.get("candidate_value")}
    if schema == "equation_solver.sweep_result.v1":
        runs = [classify(r.get("result", {}))
                for r in j.get("runs", [])]
        ok = (ok_code and j.get("all_runs_successful") is True
              and bool(runs) and all(r["accepted"] for r in runs))
        return {**base, "kind": "sweep", "accepted": ok,
                "runs": runs}
    if schema == "equation_solver.field_result.v1":
        target = j.get("accuracy_target_met")
        ok = (ok_code and j.get("field_successful") is True
              and target is not False)
        return {**base, "accepted": ok,
                "kind": "field" if target is True else "fixed_grid",
                "target_assessed": target is not None}
    if schema == "equation_solver.ode_result.v1":
        return {**base, "kind": "ivp", "accepted":
                ok_code and j.get("integration_successful") is True}
    if schema == "equation_solver.bvp_result.v1":
        return {**base, "kind": "bvp", "accepted":
                ok_code and j.get("bvp_successful") is True}
    if schema == "equation_solver.result.v1":
        ok = (ok_code and (status == "constant_satisfied" or
              (j.get("feasible") is True and bool(j.get("solutions")))))
        return {**base, "kind": "algebraic", "accepted": ok}
    return {**base, "kind": "unknown_or_error"}
```

Scope of the example

accepted denotes a family-specific completed calculation, not a global error or completeness certificate. A fixed-grid field is labelled fixed_grid with target_assessed=false. Terminal IVP events can be successful while integration_complete is false; reached_time and events must still be inspected. Algebraic acceptance says nothing about whether every root was found. Sweep child records have no invented process exit codes; nested success semantics are inspected directly.

Schema identifiers remain v1 despite additive Version 3 content. Treat unknown optional fields tolerantly, but do not coerce unknown problem types into an algebraic list. For applications requiring a particular accuracy target, additionally reject fixed_grid classifications. Validate required field types at the host boundary when processing untrusted or externally stored JSON.

Precision preserving consumption

```
from decimal import Decimal

view = classify(data, return_code)
if view["kind"] == "quadrature" and view["accepted"]:
    integral = Decimal(view["value"])
    error_text = data.get("estimated_absolute_error")
    estimated_error = (Decimal(error_text)
                       if error_text is not None else None)
```

Borrowed DLL bytes must be decoded/copied while eq_result is alive, before classifying. Free the result exactly once. Do not convert numerical strings to binary64 for storage merely because a display plot uses double. A host timeout or loader error can occur without JSON; classify only after a real response has been decoded.

33 Desktop panels numerical exports and snapshots

Version 3 adds presentation paths without moving numerical acceptance into the GUI. Review desktop/main.cpp, desktop/model.hpp, desktop/field_plot.hpp and desktop/studies_panel.hpp together when changing result interpretation.

Panel hierarchy and display precision

The main notebook has Solutions, Run report, Diagnostics, ODE / BVP / integral plot, ODE events, PDE fields / animation and Convergence / eigenpairs / sweeps. StudiesPanel contains its own Convergence, Eigenpairs and Parameter sweep tabs; Parameter sweep is not a separate top-level notebook page. Run and field selectors control the displayed child result.

FieldPlot renders numerical, exact or signed-error maps, contours and spatial sections. Animate/Pause plays computed time frames only. StudiesPanel displays grids/errors/eigenpairs and sweep summaries, overlays selected curves and exports PNG/CSV. Steady PDE comparisons use middle-y sections and transient comparisons final-time sections. Integral curves and selected modes use their sampled x coordinate.

Export details that wrappers must not assume

Generic model/result export helpers honour their documented number style and preserve engine strings. StudiesPanel::exportTable is a separate implementation: it writes quoted comma-separated cells with CRLF and has no US/EU formatting branch in this snapshot. Its sweep maximum is computed in display double precision and formatted with about 12 digits. Do not mistake that display summary for a native-precision optimisation result.

StudiesPanel sweep summaries can access computed samples even when a nested result failed its accuracy target. Any custom plot/export extension must retain per-run status and target state. Matching curves or coloured maps do not certify accuracy. A quadrature run fills the convergence table and scalar summary, not an ODE trajectory.

Host budgets and interruption

The Desktop has a 4 MiB run-model limit, 5 MiB text-reader cap and separate 64 MiB stdout/stderr buffers. The engine sample caps do not guarantee a response fits those buffers. Reduce returned density or batch size when appropriate. Forced Stop kills the worker rather than receiving final JSON, so the previous completed snapshot must remain labelled as such. A cooperative time-limit result and a host kill are different outcomes.

34 Regression design and platform acceptance

Numerical changes should begin with a minimal reproducer and an independent mathematical reference or diagnostic expectation. Cover CLI, DLL and Desktop interpretation separately; they share computations but have distinct ownership, entitlement and presentation risks.

```
python tests\version3_numerics.py engine\bin\equation_solver.exe
python tests\version3_numerics.py engine\bin\equation_solver.exe --extend
python tests\version3_dll.py engine\bin\equation_solver.dll
python tests\version3_dll.py engine\bin\equation_solver.dll --extend
python tests\quadrature.py engine\bin\equation_solver.exe
python tests\quadrature.py engine\bin\equation_solver.exe --extend
python tests\quadrature_dll.py engine\bin\equation_solver.dll
python tests\quadrature_dll.py engine\bin\equation_solver.dll --extend
```

Run from the extracted source root with an entitled configuration. Native development executables/shared objects can be supplied to the Python harnesses; such runs do not execute the Windows runtime. The supplied PowerShell Run_Version3_Checks.ps1 is an additional packaged smoke test, not the whole regression suite. Read its actual assertions before extending it.

Area	Positive reference	Adversarial or failure case
Boundaries	u=x^2+y^2 with mixed data	Sign errors, incompatible corners and Neumann balance
Refinement	Manufactured heat/integral functions	Target failure with retained samples and grid limits
Eigen	Sine modes and Neumann zero mode	Invalid weights, nonhomogeneous data and unresolved modes
Transport	Linear source chosen from a known solution	Invalid diffusion, excessive centred Peclet and coarse time resolution
Coupled Volterra	Matrix kernel with exp(x)*(1,2)	Missing entries, singular endpoint matrix
Sweep	Known stiffness scaling	Invalid per-case model, timeout and storage limits
Quadrature	pi/2, sqrt(pi), 35*pi/256 and 64/35	Divergence, interior poles, cancellation and false zero
Host routing	Accepted, syntax and fixed-grid families	Null candidate, unknown schema, absent JSON and partial batch

Separate evidence levels

The release audit records native numerical tests and static Windows PE checks. Neither proves Windows numerical-runtime behaviour, GUI event handling or activation on a user account. Run the original

endpoint-cancellation reproducer on actual Windows and capture its binary versions before marking it repaired. Never downgrade the regression merely to make an acceptance script pass.

Test the source/build combinations actually shipped: ordinary and extended precision, CLI and DLL, Commercial denial/permission boundaries, cancellation, error messages and numerical string exports. Preserve licensing and signed activation during updates. New low-impact documentation does not require rebuilding the solver, but changed public examples should at least be parsed and checked against their stated reference semantics.

35 Build identity packaging and release gates

Public release number, numerical build revision and retained Desktop file version are separate identifiers. A package can legitimately retain an unchanged compatible frontend while updating its numerical worker, provided that identity is explicit and the binaries/headers/runtime files remain consistent.

Item	Identity in this corrected candidate
VERSION.txt / product guide	3.0.1
engine/src/version.rc CLI/DLL	File/product version 3.0.1.1
Desktop binary / desktop/app.rc	Compatible retained file version 3.0.1.0
C ABI	1, ten existing exports
Windows execution	Acceptance still required; cross-compilation is not execution

Resource and installer checks

engine/build_windows.cmd invokes windres twice and links separate CLI/DLL resource objects. installer/PatodiaEquationSolver.iss guards the expected Desktop and numerical file versions and uses installer VersionInfoVersion=3.0.1.1 while AppVersion remains 3.0.1. installer/validate_payload.ps1 checks numerical FilePrivatePart=1 as well as major/minor/build, payload version, required files, private/source exclusions and manifest verification.

The supplied kit is staged source and payload; no new installer EXE was compiled in this environment. If building with CMake, address its engine version-resource omission before using its outputs with the installer guards. Verify PE32+ AMD64 architecture, imports, ten exports, subsystem and actual version resources; a renamed file is not a build identity.

Keep source documentation and all payload copies aligned

```
rebuild_engine_windows.cmd
powershell -NoProfile -ExecutionPolicy Bypass -File ^
  tools\refresh_manifest.ps1 -Root dist
powershell -NoProfile -ExecutionPolicy Bypass -File ^
  installer\validate_payload.ps1
```

The rebuild script refreshes dist but is not a complete source/archive/installer synchronisation pipeline. Stage matching engine/bin, dist and installer/payload explicitly. Copy the revised user/developer guides and examples into the correct documentation locations; the current payload validator expects docs/Equation_Solver_User_Guide.pdf. Refresh nested SHA256SUMS.txt manifests from leaves outward, then verify the archive CRC and extracted hashes. Do not claim this documentation delivery has changed an already distributed solver ZIP.

Do not ship signing private keys, customer licence files or trial state. Retain required third-party notices. Test a freshly extracted Windows folder, including spaces in paths, loader dependencies, CLI/DLL parity,

GUI solve/stop/export and licence acceptance. Keep a baseline candidate during the planned test month and record each correctness fix with its reproducer, source identity and platform results.

36 Standalone mode dispatch and isolated grammars

The engine dispatcher in engine/src/equations.tpp selects standalone modes using their leading marker. Include the associated headers through engine/src/equations.hpp, and update Desktop routing in desktop/main.cpp and desktop/model.hpp. Preserve the legacy solver path when no standalone marker is present.

```
compute / func
differentiate / diff
interpolate / interp
limit / lim
quadrature / quad
singularities
```

A pair separated by a slash denotes alternative first-line markers. singularities is a standalone marker and also names a mode-local candidate or split-point field. Quadrature accepts integrand or expr, variable or var, and interval or range for bounds. The equation declaration var x, y remains a separate grammar from var = x.

Design rule: keep Compute-only special-function identifiers and number-theory routines out of the shared equation/ODE/PDE/quadrature/differentiation/limit expression grammar unless the change is intentionally cross-cutting and exhaustively regression-tested. Functions that are already shared, including acosh, should keep existing behaviour.

37 Expression and tabular differentiation implementation

```
diff
expr = x^2*exp(y)+sin(x*z)+y^3*z
at x = 1
at y = 0
at z = 2
gradient = true
hessian = true
```

engine/src/differentiation.hpp handles standalone expression requests. wrt selects a single or mixed partial; order is 1–4. The function and first derivatives can reuse automatic differentiation; higher derivatives/Hessians rely on resolution-controlled extrapolation and report heuristic rather than certified accuracy. Coordinate validation must reject missing/nonfinite variable values and conflicting aliases.

Tabular pathway

```
diff
file = "tests/tabular_research_data.csv"
x_column = time
y_column = displacement
error_column = sigma
method = savitzky_golay
window = 7
degree = 2
at x = [3,5,7]
order = 1
uncertainty = true
output_csv = "derivative_results.csv"
```

engine/src/tabular_differentiation.hpp and tabular_research.hpp cover inline data, CSV columns and local polynomial, Savitzky–Golay and smoothing-spline estimates. Sorting irregular x values is supported; duplicated x, nonfinite measurements, inadequate samples and unsupported extrapolation must fail clearly. For splines support only derivative orders 1–2. Propagated standard uncertainty assumes independent measurement errors, not a bound on model bias. CSV output may overwrite an existing target; hosts should treat paths as user-controlled outputs.

38 Interpolation algorithms and CSV contracts

```
interpolate
data = [[0,0],[1,1],[2,4],[3,9],[4,16],[5,25]]
method = pchip
at x = [1.5,2.5]
```

engine/src/interpolation.hpp dispatches linear, local_polynomial, pchip, cubic_spline, akima, barycentric and smoothing_spline. auto selects local_polynomial. Default local window is 5 and degree 2; degree must be smaller than window, window <= 15. Akima needs at least five samples; global barycentric accepts at most 32. A smoothing spline uses lambda with 0 selecting interpolation through the points.

CSV input uses file, x_column, y_column and optional error_column. Output uses output_csv with columns x,value,standard_uncertainty. Optional uncertainty propagation supports linear/local_polynomial/barycentric/cubic_spline/smoothing_spline, not PCHIP/Akima. Extrapolate is false by default and true is accepted only for linear/local_polynomial. Keep Desktop reporting and CSV export tested independently of numerical values.

39 Numerical limits and evidence semantics

```
limit
expr = (exp(x)-1)/sqrt(1-cos(x))
at x = 0
direction = both
```

engine/src/limits.hpp handles finite and ±infinite target coordinates and left/right/both classification. This example is does_not_exist: the left side tends to -sqrt(2), the right to sqrt(2). In particular, $\sqrt{x^2}$ equals abs(x) over the reals, so cancellation and radical rewrites must remain side-aware. A scalar return value is inappropriate when the sides disagree.

Result classifications are finite_limit, positive_infinity, negative_infinity, does_not_exist and undetermined. Preserve method/evidence, evaluation precision, heuristic absolute error, requested tolerance and optional sided values in JSON/desktop. Do not convert undetermined into does_not_exist merely because a sampler stopped. For multivariable paths, conflicting approaches disprove a limit; agreeing finite samples are not a proof. The current numerical/structural evidence is not formal certification. The updated limit evaluator improves numerical handling. Preserve sided evidence and the distinction between conflicting approaches and insufficient evidence.

40 Compute Functions expression engine and explorer

```
compute
precision = binary128
a = gamma(5)
b = erf(1)
c = besselj(0,2)
d = lambertw(5)
e = zeta(2)
total = a+b+c+d+e
```

Compute is an independent grammar implemented through engine/src/compute_functions.hpp and compute_high_precision.hpp. Named scalar assignments execute sequentially and may reference earlier assignments. For a single result expr = gamma(5) is accepted. Avoid silently admitting Compute-only functions into legacy parsers.

```
compute
function = besselj(0,x)
range = [0,20]
points = 500
```

Explorer samples endpoints inclusively, accepts 2–20,000 points and returns per-sample x,y,status rows. An invalid sample can yield a null y; fail the complete explorer request if no sample is finite. Desktop tables/plots and CSV export need separate regression coverage. The schema is equation_solver.compute.v1, with mode calculations or explorer and certified=false.

Catalogue categories include elementary trigonometric/hyperbolic, gamma/error, Bessel, Airy, orthogonal polynomials, zeta, Lambert W, exponential/sine/cosine integrals and elliptic integrals. Maintain a single canonical function-name/arity/domain registry in Compute and reject unknown names or wrong argument counts rather than returning plausible zeros.

41 Boost Multiprecision and precision boundaries

The supplied backend uses Boost.Math 1.83 and Boost.Multiprecision cpp_bin_float_quad: a 113-bit significand (about 34 decimal digits). This is binary128-class precision, not an assurance of 34 correct digits on every ill-conditioned special-function input. precision = double selects the compatibility path. Parse decimal literals directly into the target backend; do not round via a double first.

Required directory and Windows build

```
engine\boost_full\boost\multiprecision\cpp_bin_float.hpp
engine\boost_full\boost\math
engine\boost_full\boost\config

engine\build_windows.cmd
```

Extract the complete Boost 1.83 boost directory, including all transitive header dependencies. The portable header implementation does not require precompiled Boost libraries, but the Windows engine build still links and deploys its other required runtime libraries. Confirm MinGW-w64 x64 GCC and windres are available and check both CLI and DLL output. Source-level Linux success does not establish a Windows binary build or MSIX behaviour.

42 Compute number theory and exact integer outputs

```
compute
precision = binary128
a = totient(5929)
b = primepi(100000000)
c = primefactors(5929)
d = primefactor(5929,3)
```

Compute-only exact-integer functions are totient(n), gcd(a,b), lcm(a,b), isprime(n), primepi(n), mobius(n), divisorcount(n), divisorsum(n), primefactors(n), primefactor(n,k). Strict upper bounds are n <= 100,000,000 for primepi, inclusive, and input values <= 10^12 for the other number-theory routines. primepi(100000000) = 5761455. Invalid integers or values beyond configured ceilings must be rejected before work begins.

primefactors returns a structured list of decimal integer strings, sorted with multiplicity; it is not a scalar and must not become a variable for later arithmetic. primefactor(n,k) returns one 1-based factor. Report exact integers as strings in JSON to protect downstream JavaScript clients from precision loss. Keep bounded factorisation resource costs and invalid-argument regressions.

43 Exact linear Diophantine classification

```
int x, y
7*x + 11*y = 100
```

engine/src/linear_diophantine.hpp recognizes exactly one linear equality in two integer variables after exact polynomial normalization. Using gcd and the Extended Euclidean Algorithm, test divisibility of c by gcd(|a|,|b|), then produce the complete family x=x0+(b/g)t, y=y0-(a/g)t for integer t. For this example x=8+11t, y=4-7t, equivalently $x \equiv 8 \pmod{11}$ together with y=(100-7x)/11.

Intersect integer-parameter constraints from positivity and explicit bounds. Classify none, finite (exact count), or infinite without confusing the number of enumerated display rows with mathematical completeness. Handle zero and signed coefficients, degenerate 0=0 / 0=c, large integer coefficients, and algebraically rearranged forms. JSON additions include problem_type=linear_diophantine, solution_cardinality, solution_count, general_solution, modular_solution and runtime_nanoseconds. Human time formatting should not round very fast exact solves to an uninformative 0 s.

44 Windows MSIX assembly and release verification

One source baseline and one binary set

Freeze one source revision; rebuild engine/bin/equation_solver.exe, engine/bin/equation_solver.dll and Desktop from that same revision. Confirm every runtime DLL is present in the actual package layout. Synchronise distribution folders and installer payload copies, retaining correct architecture and Windows version resources. Keep the third-party licence notices; do not include signing keys, live licence state, test secrets or intermediate object files.

Windows acceptance matrix

1. Run tests/run_differentiation_regression.py, run_tabular_differentiation_regression.py and run_tabular_research_regression.py against the rebuilt Windows CLI.

2. Run run_interpolation_regression.py, run_limits_regression.py, run_compute_functions_regression.py, run_compute_number_theory_regression.py and run_linear_diophantine_regression.py.

3. Run the existing equation, integer/Pell, quadrature, IVP/BVP, PDE/eigen, integral-equation and desktop result-format suites; execute the chapter-level challenging models. Include the range-splitting, singularity, limit and Desktop scenarios in Chapter 45.

4. Test CLI JSON, DLL ABI, Desktop result pages, graphs, CSV exports, cancellation and malformed inputs at double and high precision where supported.

5. Install the MSIX on a clean Windows x64 account, launch the packaged Desktop, verify worker and DLL discovery, licence/trial behaviour, file access, uninstall and upgrade.

6. Inspect package manifest publisher identity, package version, declared capabilities, signing certificate, dependency declarations and Microsoft Store validation results.

7. Regenerate SHA256SUMS.txt, START_HERE.txt, VALIDATION.md and package inventories only after files are final; independently check hashes against the final payload.

The above is a release acceptance checklist, not a statement that the current MSIX already passed. Record command lines, toolchain versions, test counts, failures and the package SHA-256. Do not advertise the new features as shipped until the final installed package passes those checks.

45 Quadrature ranges singularity analysis and Desktop tools

Quadrature model forms and acceptance

Treat the explicit range list and the single range with declared singularities as two separate model forms. The first integrates the same expression on each supplied interval and combines their contributions. The second uses the supplied singularities as split points in the overall range. Run each model independently in US number style.

```
quad
expr = 1/sqrt(abs(x-0.2))
var = x
range = [0, 0.2], [0.2, 1]

quad
expr = 1/sqrt(abs(x-0.2))
var = x
range = [0, 1]
singularities = [0.2]
```

Both models have reference $6/\sqrt{5} \approx 2.683281573$. Assert accepted status and agreement with the reference within the test tolerance. Also exercise malformed range syntax and an interior nonintegrable pole. Confirm that an interrupted or unaccepted run cannot surface an accepted scalar through the host.

Standalone singularity analysis

```
singularities
expr = x * asin(2x * sqrt(1-x^2)) / sqrt(1-x^2)
var = x
range = [-1, 1]
// singularities = [-1/sqrt(2), 1/sqrt(2)]
```

The optional field supplies candidate points when uncommented. Exercise both forms. The function is finite with corners at $\pm 1/\sqrt{2}$, where the arcsine argument reaches ±1. At ±1 the formula is undefined and the one-sided limits from the interior are both 2. These cases exercise distinct local behaviours. Preserve the returned evidence and distinguish analysis of a candidate from acceptance of a quadrature result.

Limits and bounded validation

Exercise finite one-sided limits, differing left and right limits, and cases with insufficient evidence. Keep the classifications in Chapter 39. For ODEs, BVPs, PDEs and integral equations, test that bounded numerical investigation can add validation evidence and that unresolved or ambiguous outcomes retain their diagnostic status. Check Desktop and any supported host interfaces against the final run report.

Desktop regression cases

Tools → Equation Generator supports 1–20 variables, selectable function families and complexity, optional bounds, and seed-based reproduction. Verify repeated generation with the same seed and settings, substitute the constructed solution into the emitted equations, and check that exact simplification preserves the identities. Cover combining like terms and powers, common factoring when shorter, and removing redundant parentheses. Verify that the known solution is revealed on request.

Exercise each Fun with Mathematics activity in the installed Desktop. Test digit-sequence lookup in Find Your Number in Pi, the smallest starting number under successive divisibility conditions in The King's Puzzle, and tile arrangements against the arithmetic rules in Number Puzzle.

46 Workflow execution and service dispatch

Write within <seconds> for a per-job budget. The word in belongs to a loop header such as for i in 1..10. A service block starts with its keyword and a unique result name in the current block. Inside it, use the standalone service's directives without repeating its keyword. Close the block with end.

The eight headers shown with within enforce an optional per-job budget. In the workflow timing update, compute, diff, limit, and interpolate accept a valid within clause, run normally and record an informational notice that the cap was ignored. The same rule applies to reusable models.

Service	Header example	Where to find the result
Equation solving	solve roots within 2	roots.roots, roots.status
Definite quadrature	quad area within 2	area.value, area.estimated_absolute_error
Compute Functions	compute calc	calc.result.results for calculations
Differentiation	diff slope	slope.result.requested_derivative when requested
Limit	limit near_zero	near_zero.status, near_zero.value for a finite value
Interpolation	interpolate sample	sample.result.results
ODE initial-value problem	ode decay within 2	decay.result.trajectory
Boundary-value problem	bvp curve within 2	curve.result.trajectory
PDE	pde square within 2	square.result.trajectory
Eigenvalue problem	eigen modes_found within 2	modes_found.result.eigenpairs
Integral equation	integral fredholm within 2	fredholm.result.trajectory
Singularity analysis	singularities poles within 2	poles.result.points

Script structure and reusable models

Define a model at the top level with model name(parameters) as service and end. Use @parameter in the service body to insert a numeric value. Call it with run result = name(arguments). A supported service can also take run result = name(arguments) within seconds. Pass every model parameter explicitly. For a direct job, @name or @root.x reads a numeric value from the enclosing workflow. Declare solver bounds inside the solve body.

For exact integer equations, write large exact literals directly in the service body. Numeric workflow arithmetic uses the selected engine precision. Direct substitutions preserve stored decimal text until you apply arithmetic.

47 CLI and native DLL workflow integration

Command line and validation

From Command Prompt, run the following with the executable on your PATH or from its installation directory.

```
equation_solver.exe --file workflow.txt --seconds 20
equation_solver.exe --file workflow.txt --seconds 20 --json
equation_solver.exe --file workflow.txt --check
```

Add --extend to select extended precision where your licence and service support it. For a DLL client, pass the same workflow text as the model to eq_solve. Pass options such as --seconds and 20 as separate tokens, then release the result using eq_result_free. The DLL requires a DLL-enabled entitlement.

--check validates workflow structure and checks service models that can be checked without runtime substitution. A model with @parameters is validated when it runs. Run the workflow to verify time-cap support, runtime names and parameter values.

CLI exit code 0 means complete without unaccepted jobs, or syntax check passed. 2 means at least one unaccepted job or a resource stop. 1 indicates a syntax or execution error. In JSON, check the top-level status and the individual jobs and reports.

Native host lifecycle

Pass the entire UTF-8 script as the model to eq_solve using the declaration from the matching public header. Keep the model and option strings valid during the call. Inspect the returned status and result before calling eq_result_free. Apply the lifetime and ownership rules in Section 9.

Result routing and status handling

job.status is the service status. job.accepted is a Boolean that reflects the workflow's acceptance rule for that service. A successful solve can return roots even when a full search has not been established.

job.value is available for accepted scalar results when the service returns a value. A finite limit has a scalar value; an accepted infinite limit is described by its status. Use job.result for the detailed response of the services other than solve and quad. For those two, the full response is still present in Run Report and the CLI's jobs[].result JSON.

Treat workflow_complete_with_job_failures as a batch with unaccepted jobs. Read the recorded jobs and reports after time_limit or step_limit.

48 Workflow budgets and regression checks

Whole workflow. Set Compute → Solver Settings → Time budget, seconds in Desktop or --seconds S in the CLI. The CLI defaults to 20 seconds. A value of 0 disables the overall time budget.

Individual job. within takes a positive finite number or a numerical expression such as within budget/2. A supported job receives the smaller of its cap and the remaining workflow budget. If the overall budget is disabled, the explicit job cap still applies.

Timing precision. A budget is checked cooperatively. The workflow checks it between statements and jobs. Services without per-job support may finish their current operation before the workflow can observe that its budget expired. A capped job can also overrun briefly between internal checks. Give a batch enough total time for the jobs and workflow overhead.

Job failure versus script error. A service failure can be recorded and the script can continue while budget remains. A malformed or nonpositive within value or an invalid workflow expression stops the script with an error. An exhausted overall budget stops the batch.

Workflow limits include 4 MiB of source text, 10,000 input entries, 32 nested blocks, 32 active function calls, 1,000 jobs, 10,000,000 execution steps, and 100,000 items in a range or collection. Range endpoints must lie within ±1 billion. The step budget is shared across the entire run. step_limit preserves reports and jobs recorded before the stop. Recorded results are limited to 8 MiB and tracked collection copying and appending to 32 MiB. For large sweeps, split the run into batches.

Timing and failure regression

This example gives quadrature a cap and runs differentiation using the overall workflow budget.

```
workflow
budget = 2
quad area within budget
    expr = x^2
    var = x
    range = [0, 3]
end
diff slope within 2
    expr = x^3
    at x = 2
    wrt = x
end
if area.accepted
    report area.value
else
    report area.status
end
report slope.result.requested_derivative
```

Expected reports are 9 and 12, plus one notice about the ignored differentiation cap.

In the target Windows package, compare CLI text, CLI JSON, DLL results and the Desktop Run Report. Exercise a malformed cap, an unaccepted job followed by a successful job, a timeout and a step-limit stop. Verify the top-level status, job order, retained reports and process exit code where applicable.