Patodia Equation Solver 4.0 Workflow Guide

Use workflows to automate numerical studies, sweep parameters, combine solver services and postprocess results in a single script. This guide covers the Windows x64 4.0 update with a 10,000,000-step budget and optional legacy `while` caps.

Patodia Infotech Private Limited

Contents

- Run a workflow
- Time budgets and the 12 services
- Variables conditions and loops
- User defined functions and while loops
- Reusable models parameter sweeps and collections
- Read results and troubleshoot
- Worked programming examples
- Function reference
- Copy and run service examples

Run a workflow

Start the file with `workflow`. In Windows Desktop, open the file, choose **Compute → Check Syntax** if you want a preliminary check, then choose **Compute → Solve**. Read **Run Report** for each job's full output. The workflow table shows reports and job statuses, including the loop inputs associated with each row.

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.

Time budgets and the 12 services

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`.

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`

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.

- **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.

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.

Estimate the overall budget

Run Check Syntax and look in **Diagnostics** for timing guidance. In the CLI, use `--check` or inspect `timing` in JSON. In a completed or time-limited run, the guidance appears in **Run Report**. Each JSON job has elapsed time, its requested cap, and its effective budget.

- A fixed literal range or list can contribute a pre-run cap total. Ten jobs with `within 2` contribute 20 seconds of configured allowance. The report flags an overall budget below that allowance. Allow extra time for overhead and for uncapped jobs if every capped job should receive its full allowance.
- Variable-dependent counts or caps and jobs under `if` or `while` may leave the pre-run total partial or unknown. The run records the configured caps of jobs it reaches, excluding ignored caps.
- A successful measured run can suggest a budget for a similar run. After an overall timeout, the workflow can extrapolate from completed jobs at the same workflow statements when it knows the remaining job counts and has matching samples. Suggestions add a 25% margin and round up to a whole second. Input difficulty, machine load and solver behavior can change the runtime.
- After an overall timeout, read the count of recorded job results and the last active statement. When a job reaches its own cap, review that job separately. Raising the overall budget alone cannot extend an individual cap.

Variables conditions and loops

Construct	Meaning
`x = expression`	Update the nearest existing `x`, or create it in the current block.
`let x = expression`	Declare a local name in the current block.
`set x = expression`	Update an existing binding.
`x += expression`	Add to an existing numerical binding.
`if condition` … `else` … `end`	Run one branch. `else` is optional.
`for i in first..last` … `end`	Iterate inclusive integer endpoints in ascending or descending order.
`for item in items` … `end`	Iterate a snapshot of a collection or a list literal.

Construct	Meaning
`while condition … end`	Repeat while true. The workflow-wide limits apply.
`break`	Exit the innermost loop.
`continue`	Move to the next iteration of the innermost loop.
`report expression`	Add a result row with the enclosing `for` case values.

Conditions accept Booleans or a comparison using `==`, `!=`, `<`, `<=`, `>` or `>=`. Compare textual statuses with quoted strings, such as `if area.status == "integration_complete"`. Use explicit multiplication in workflow expressions, such as `2*x`. Loop iterations and branches have local scope. Initialize accumulators outside the loop so updates persist.

```
workflow
sum = 0
for i in 1..10
    sum += 1/i
end
report sum
```

Expected result is approximately `2.928968253968254`.

Expressions and scope

Write one workflow statement per line, start with `workflow`, and put each closing `end` on its own line. Indent the body for readability. Use `//` for line comments. Names start with a letter or underscore and may contain digits after the first character. Use lowercase keywords and descriptive names such as `root_count`. Choose names distinct from service keywords and constants.

Element	Syntax and meaning
Numeric values	`-2.5`, `1e-12`, `#pi`, `#e`. Use explicit constants in portable scripts.
Arithmetic	`+`, `-`, `*`, `/`, `%`, `^` or `**`. Parentheses group expressions.
Text and Booleans	`"finite_limit"`, `true` and `false`.
Fields	`root.x` or `decay.result.trajectory`. Each dot selects a field in a record.
Collections	`[]`, `[1, 2, 3]`, or `[[1,2], [3,4]]`. Use `at(items, 1)` to read the first item.
Number style	US uses `at([1.5, 2.5], 2)`. EU uses `at([1,5; 2,5]; 2)`. Set the style before loading or running examples.

A condition can be a Boolean field such as `area.accepted` or a single comparison. For compound tests, nest `if` blocks. An assignment without `let` updates the nearest existing binding. Use `let` for a fresh local binding, and use `set` when a missing binding should raise an error. Variables declared inside a loop or branch disappear when that block ends. Define accumulators before entering the loop.

Filter and exit a loop

This example visits a range in descending order. It skips even integers and stops after reporting an odd integer less than 5.

```
workflow
for n in 9..1
    if n % 2 == 0
        continue
    end
    report n
    if n < 5
        break
    end
end
```

The reports are `9`, `7`, `5` and `3`. For nested loops, `break` and `continue` act on the innermost loop. The enclosing `for` values appear in each report's case fields.

User defined functions and while loops

Define a numerical function at the top level with `function name(parameters) = expression`. It uses its explicit numerical parameters and can call other workflow functions, including functions defined later in the script. Each invocation sees only its own parameters. Pass values from the surrounding workflow as arguments. The engine allows up to 32 active calls.

A `while` loop rechecks its condition at the start of each iteration. It ends when the condition is false, a `break` exits it, or the workflow stops at a resource limit. Older scripts with an explicit iteration cap continue to honour it.

```
workflow
function improve(x) = (x + 2/x)/2
x = 1
iterations = 0
while abs(x*x - 2) > 1e-12
    x = improve(x)
    iterations += 1
end
report x
report iterations
```

Expected results are `1.414213562373095` and `5`.

Reusable models parameter sweeps and collections

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.

Use `[]` or a list literal to create a collection. `append name with expression` adds a value. US syntax also accepts `append name, expression`. `count(name)` counts entries, and `at(name, index)` uses indices starting at 1. A collection can contain numbers, records, or other collections. Store results in a collection initialized outside the loop to retain them across iterations.

```
workflow
model power_integral(b) as quad
    expr = x^2
    var = x
    range = [0, @b]
end
values = []
for n in 1..3
    run area = power_integral(n) within 2
    if area.accepted
        append values with area.value
    else
        report area.status
    end
end
report values
report count(values)
report at(values, 2)
```

Expected values are approximately `[0.3333333333333333, 2.6666666666666667, 9]`, count `3`, and second entry `2.6666666666666667`. In EU number style, use decimal commas and semicolons inside lists, intervals and function arguments, such as `at(values; 2)`.

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.

```
workflow
for p in 50..53
    solve pair within 2
        pos int x, y
        x^2 + @p*y^2 = 81755817846740768162132860 86
    end
    report p
    report pair.status
    report count(pair.roots)
end
```

Expected counts for `p = 50, 51, 52, 53` are `7, 0, 0, 6`. Expect `workflow_complete_with_job_failures` because the empty solves are recorded as unaccepted jobs. For exact enumeration, consult the job's stopping reason and integer-completeness flags in the full JSON. The generic `search_complete` field describes a separate certification scope.

Pass a result to another service

Read a root field in the workflow and use `@root.x` to pass its numeric value to a job. Inside a service block, write that service's model syntax. Use ordinary workflow names outside service blocks. A reusable model sees only its declared parameters for substitution.

```
workflow
solve scale
    x = 2
end
for root in scale.roots
    quad area
        expr = @root.x*t
        var = t
        range = [0,1]
    end
    if area.accepted
        report area.value
    else
        report area.status
    end
end
```

Expected area is `1`. Open the two jobs in Run Report to inspect the solve and integration independently.

Read results and troubleshoot

- `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.
- `quad` requires an ordinary integral. A pole may make it divergent even when a Cauchy principal value exists. Run `singularities` to investigate the interval and inspect the job's reason when a result is unaccepted.
- `--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`.
- 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.

Result fields and failure handling

Field	Use
`job.status`	Inspect the outcome reported by the service.
`job.accepted`	Guard the use of a result. This field uses the workflow's service-specific acceptance rules.
`job.value`	Read an accepted scalar when the service supplies one. For `diff`, use the specific requested derivative field.

Field	Use
`job.roots`	Iterate the solution records returned by `solve`. Access each variable by name, for example `root.x`.
`job.search_complete`	Inspect the engine's general search-completeness flag. Read the full response for its scope and any exact-integer completeness fields.
`job.estimated_absolute_error`	Read the service estimate when present.
`job.result`	Inspect the detailed result for services other than `solve` and `quad`.
`jobs[].result`	Inspect every full service response in CLI or DLL JSON.
`reports[]` and `timing`	Read reported expressions, values and loop cases, plus workflow timing diagnostics.

Top level status	What to check
`syntax_valid`	The syntax pass completed. Run the script to evaluate dynamic inputs.
`workflow_complete`	Review the reports and inspect service diagnostics for accuracy or completeness.
`workflow_complete_with_job_failures`	Inspect each unaccepted job's status and reason.
`error`	Read the workflow statement and error message. Check bindings, fields, number style and model inputs.
`time_limit` or `step_limit`	Review recorded output. The time budget measures elapsed time. The step budget counts workflow operations and loop checks.
`job_limit`, `collection_limit` or `output_limit`	Reduce the number of jobs, retained collections or reported values in one run.
`while_limit`	An older script used an explicit iteration cap and exhausted it.
`cancelled`	The engine received a cancellation request.

Worked programming examples

Sum the first 10000 primes

The script seeds the result with 2, then checks only odd candidates. It uses the scalar `isprime` function directly in workflow expressions.

```
workflow
// Sum the first 10,000 primes. Expected: 10000, 104729, 496165411.
total = 2
prime_count = 1
last = 2
candidate = 1

while prime_count < 10000
    candidate += 2
    if isprime(candidate) == 0
        continue
    end
    total += candidate
    prime_count += 1
    last = candidate
end

report prime_count
report last
report total
```

Expected count, last prime and sum are `10000`, `104729` and `496165411`.

Find a birthday probability threshold

Assume independent birthdays distributed uniformly over 365 days. Keep multiplying the probability that all birthdays are distinct until the complementary probability reaches one half.

```
workflow
people = 0
distinct = 1
while 1 - distinct < 0.5
    distinct = distinct * (365 - people)/365
    people += 1
end
report people
report 1 - distinct
```

Expected results are `23` and approximately `0.5072972343`.

Use one parameter across numerical services

The workflow solves for a coefficient, then uses it in a definite integral, an ODE, a BVP and a Poisson PDE.

```
workflow
// Solve for a shared parameter and use it in four numerical services.
solve parameter
    x = 2
end

for root in parameter.roots
    a = root.x
    quad area
        expr = @a*x
        var = x
        range = [0,1]
    end
    report area.value

    ode decay
        y' = -@a*y
        initial t = 0
        initial y = 1
        interval t = [0,1]
        output points = 11
    end
    report decay.status

    bvp beam
        y'' = -@a
        interval x = [0,1]
        boundary left y = 0
        boundary right y = 0
        output step = 0.1
    end
    report beam.status

    pde plate
        type = poisson
        u_xx+u_yy = -@a*#pi^2*sin(#pi*x)*sin(#pi*y)
        domain x = [0,1]
        domain y = [0,1]
        boundary left u = 0
        boundary right u = 0
        boundary bottom u = 0
        boundary top u = 0
        grid x = 17
        grid y = 17
    end
    report plate.status
end
```

The accepted integral is `1`. The remaining reports show the statuses of the ODE, BVP and PDE jobs. Read each full numerical result in Run Report.

Function reference

Workflow expressions can call the shared core functions and scalar Compute functions directly. A job body uses its own service's expression rules. For a Compute-only scalar needed by another service, evaluate it in the workflow and substitute the resulting number with `@name`. Workflow arithmetic uses the selected engine

precision. Scalar Compute functions use the high-precision Compute backend, with ordinary surrounding workflow arithmetic evaluated in the selected workflow precision. All trigonometric arguments are in radians.

The catalogue gives sample calls and a short description of each function. Check domain restrictions before applying a function to measured data or a sequence of solver outputs.

Collection function	Result
`count(items)`	Number of entries in a list.
`at(items, index)`	Value at an integer index starting at 1.
`primefactors(n)`	Compute calculation that returns a formatted factor list. Call it as a whole assignment inside a named `compute` job and read the row's `value`.

```
workflow
compute factor_list
    factors = primefactors(5929)
end
for row in factor_list.result.results
    report row.value
end
report primefactor(5929, 3)
```

The Compute job returns the formatted text `"[7, 7, 11, 11]"` for the factor list. The scalar `primefactor` call returns `11`. To build a workflow list for further processing, call `primefactor` for each index and append the result.

Shared core functions

Syntax	Meaning and domain	Example
`sin(x)`	Sine, radians. Domain All real x	`sin(#pi/6)`
`cos(x)`	Cosine, radians. Domain All real x	`cos(#pi)`
`tan(x)`	Tangent. Domain cos(x) ≠ 0	`tan(#pi/4)`
`cot(x)`	Cotangent. Domain sin(x) ≠ 0	`cot(#pi/4)`
`sec(x)`	Secant. Domain cos(x) ≠ 0	`sec(0)`
`csc(x)`	Cosecant. Domain sin(x) ≠ 0	`csc(#pi/2)`
`asin(x)`	Principal inverse sine. Domain −1 ≤ x ≤ 1	`asin(0.5)`
`acos(x)`	Principal inverse cosine. Domain −1 ≤ x ≤ 1	`acos(0)`
`atan(x)`	Principal inverse tangent. Domain All real x	`atan(1)`
`acot(x)`	Principal inverse cotangent, π/2 − atan(x). Domain All real x	`acot(0)`
`asec(x)`	acos(1/x). Domain |x| ≥ 1	`asec(2)`
`acsc(x)`	asin(1/x). Domain |x| ≥ 1	`acsc(2)`
`atan2(y,x)`	Quadrant-aware angle. Domain (x,y) ≠ (0,0); shared evaluator rejects the origin; do not rely on backend-specific values there	`atan2(1,1)`
`sinh(x)`	Hyperbolic sine. Domain All real x	`sinh(1)`
`cosh(x)`	Hyperbolic cosine. Domain All real x	`cosh(1)`
`tanh(x)`	Hyperbolic tangent. Domain All real x	`tanh(1)`
`coth(x)`	Hyperbolic cotangent. Domain x ≠ 0	`coth(1)`

Syntax	Meaning and domain	Example
sech(x)	Hyperbolic secant. Domain All real x	sech(0)
csch(x)	Hyperbolic cosecant. Domain x ≠ 0	csch(1)
asinh(x)	Inverse hyperbolic sine. Domain All real x	asinh(2)
acosh(x)	Inverse hyperbolic cosine. Domain x ≥ 1	acosh(2)
atanh(x)	Inverse hyperbolic tangent. Domain −1 < x < 1	atanh(0.5)
acoth(x)	Inverse hyperbolic cotangent. Domain |x| > 1	acoth(2)
asech(x)	Inverse hyperbolic secant. Domain 0 < x ≤ 1	asech(0.5)
acsch(x)	Inverse hyperbolic cosecant. Domain x ≠ 0	acsch(2)
exp(x)	Exponential e^x. Domain All real x; finite output only	exp(1)
expm1(x)	Numerically stable exp(x) − 1. Domain All real x; finite output only	expm1(1e-20)
log(x), ln(x)	Natural logarithm. Domain x > 0	log(#e)
log10(x)	Base-10 logarithm. Domain x > 0	log10(100)
log1p(x)	Numerically stable log(1+x). Domain x > −1	log1p(1e-20)
sqrt(x)	Principal square root. Domain x ≥ 0	sqrt(2)
cbrt(x)	Real cube root. Domain All real x	cbrt(-8)
pow(x,y)	Real power x^y. Domain x>0 for arbitrary real y; negative x requires an integer exponent in the real power evaluator; use cbrt or Compute-only root for supported real roots	pow(2,10)
abs(x)	Absolute value. Domain All real x	abs(-3.2)
hypot(x,y)	Euclidean norm √(x²+y²). Domain All real x,y; finite output only	hypot(3,4)
min(a,b)	Minimum of two values. Domain All real a,b	min(3,5)
max(a,b)	Maximum of two values. Domain All real a,b	max(3,5)

Scalar Compute functions

Syntax	Meaning and domain	Example
logbase(x,b), logb(x,b)	Logarithm in base b. Domain x > 0, b > 0, b ≠ 1	logbase(8,2)
root(x,n)	Real n-th root. Domain n ≠ 0; negative x requires odd integer n	root(-27,3)
sign(x)	Sign: −1, 0 or +1. Domain All real x	sign(-4)
floor(x)	Greatest integer ≤ x. Domain All real x	floor(-1.2)
ceil(x)	Least integer ≥ x. Domain All real x	ceil(-1.2)
round(x)	Nearest integer; half values away from zero. Domain All real x	round(-2.5)
trunc(x)	Truncate fractional part toward zero. Domain All real x	trunc(-3.8)
frac(x)	Fractional part x − floor(x). Domain All real x; result in [0,1)	frac(-1.25)
factorial(n), fact(n)	Product 1×2×…×n (0! = 1). Domain Integer 0 ≤ n ≤ 2000	factorial(6)

Syntax	Meaning and domain	Example
`gamma(x)`	Euler gamma Γ(x). Domain Real x except nonpositive integers; finite result	`gamma(0.5)`
`lgamma(x)`	Logarithm of |Γ(x)| under Boost.Math convention. Domain Real x except nonpositive integers	`lgamma(5)`
`beta(a,b)`	Euler beta B(a,b). Domain Common regular case a>0,b>0; other real values depend on gamma poles and Boost implementation	`beta(2,3)`
`erf(x)`	Gaussian error function. Domain All real x	`erf(1)`
`erfc(x)`	Complementary error function 1−erf(x). Domain All real x	`erfc(1)`
`digamma(x)`	Logarithmic derivative Γ′(x)/Γ(x). Domain Real x except nonpositive integers	`digamma(1)`
`polygamma(n,x)`	n-th derivative of digamma. Domain Integer 0 ≤ n ≤ 8; x>0 recommended (required for n>0 per documented service contract)	`polygamma(1,1)`
`besselj(v,x)`	Bessel function J of order v. Domain x≥0 for noninteger v; real-valued result required	`besselj(0,2)`
`bessely(v,x)`	Bessel function Y of order v. Domain x>0 (real branch)	`bessely(0,2)`
`besseli(v,x)`	Modified Bessel I of order v. Domain x≥0 for general real v; real result required	`besseli(0,2)`
`besselk(v,x)`	Modified Bessel K of order v. Domain x>0	`besselk(0,2)`
`airy_ai(x)`	Airy Ai solution. Domain All real x	`airy_ai(0)`
`airy_bi(x)`	Airy Bi solution. Domain All real x	`airy_bi(0)`
`legendre(n,x)`	Legendre polynomial $P_n(x)$. Domain Integer n≥0 (≤10000); real x	`legendre(3,0.5)`
`assoc_legendre(n,m,x)`	Associated Legendre polynomial. Domain Integers n,m≥0; m≤n; typically |x|≤1	`assoc_legendre(3,1,0.5)`
`hermite(n,x)`	Physicists' Hermite polynomial. Domain Integer n≥0 (≤10000); real x	`hermite(3,2)`
`laguerre(n,x)`	Laguerre polynomial. Domain Integer n≥0 (≤10000); real x	`laguerre(3,2)`
`assoc_laguerre(n,α,x)`	Generalized Laguerre polynomial (implemented integer α). Domain Integers n,α≥0 (≤10000); real x	`assoc_laguerre(3,2,1)`
`chebyshev_t(n,x)`	Chebyshev polynomial $T_n(x)$	`chebyshev_t(4,0.5)`
`chebyshev_u(n,x)`	Chebyshev polynomial $U_n(x)$. Domain Integer n≥0 (≤10000); real x	`chebyshev_u(4,0.5)`
`zeta(s)`	Riemann zeta ζ(s). Domain Real s ≠ 1; finite real result	`zeta(2)`
`lambertw(x)`	Principal real branch $W_0(x)$, W e^W = x. Domain x ≥ −1/e	`lambertw(5)`
`lambertw(x,branch)`	Real branch W_0 or W_{-1}. Domain branch integer 0 or −1; W_{-1} on −1/e ≤ x < 0	`lambertw(-0.1,-1)`
`ei(x)`	Exponential integral Ei(x). Domain Real x ≠ 0 (principal real value)	`ei(1)`
`si(x)`	Sine integral $\int_0^x \sin(t)/t\,dt$. Domain All real x; series/quadrature/asymptotic approximations	`si(1)`

Syntax	Meaning and domain	Example
`ci(x)`	Cosine integral $\gamma+\ln(x)+\int_0^x(\cos(t)-1)/t\,dt$. Domain $x > 0$	`ci(1)`
`elliptic_k(m)`	Complete elliptic integral K with parameter $m=k^2$. Domain $0\le m<1$ (real finite principal branch)	`elliptic_k(0.5)`
`elliptic_e(m)`	Complete elliptic integral E with parameter $m=k^2$. Domain $0\le m\le 1$	`elliptic_e(0.5)`
`elliptic_f(phi,m)`	Incomplete elliptic integral of first kind. Domain $0\le m\le 1$; avoid singular integration paths	`elliptic_f(1,0.5)`
`elliptic_einc(phi,m)`	Incomplete elliptic integral of second kind. Domain $0\le m\le 1$ for real modulus	`elliptic_einc(1,0.5)`
`elliptic_pi(n,m)`	Complete elliptic integral of third kind. Domain $0\le m<1$; $n<1$ for regular real integral	`elliptic_pi(0.2,0.5)`
`elliptic_pi(phi,n,m)`	Incomplete elliptic integral of third kind. Domain $0\le m\le 1$; $1-n\sin^2(t)$ must not vanish on integration path	`elliptic_pi(1,0.2,0.5)`
`totient(n)`	Euler's totient $\phi(n)$: number of integers from 1 to n relatively prime to n. $1 \le n \le 10^{12}$.	`totient(5929) = 4620`
`gcd(a,b)`	Greatest common divisor of nonnegative integers. $0 \le a,b \le 10^{12}$; gcd(0,0)=0.	`gcd(42,77) = 7`
`lcm(a,b)`	Least common multiple of nonnegative integers. $0 \le a,b \le 10^{12}$; lcm(0,b)=0.	`lcm(14,22) = 154`
`isprime(n)`	Primality predicate: numeric 1 if n is prime, otherwise 0. $0 \le n \le 10^{12}$.	`isprime(97) = 1`
`primepi(n)`	Exact number of primes $p \le n$. $0 \le n \le 100{,}000{,}000$ (inclusive).	`primepi(1000000) = 78498`
`mobius(n)`	Möbius $\mu(n)$: 0 for repeated prime factors, otherwise (-1) to the count of distinct factors. $1 \le n \le 10^{12}$.	`mobius(30) = -1`
`divisorcount(n)`	Number $\tau(n)$ of positive divisors of n. $1 \le n \le 10^{12}$.	`divisorcount(360) = 24`
`divisorsum(n)`	Sum $\sigma(n)$ of positive divisors of n. $1 \le n \le 10^{12}$.	`divisorsum(28) = 56`
`primefactor(n,k)`	Select 1-based kth prime factor including multiplicity. $2 \le n \le 10^{12}$; $1 \le k \le 64$ and k must exist.	`primefactor(5929,3) = 11`

Copy and run service examples

Each block is a separate workflow. Use US number style. For jobs that return trajectories or other structured data, use Run Report or inspect the corresponding JSON result.

1 Algebraic equations

```
workflow
solve linear_pair within 1
    x+y = 5
    x-y = 1
end
for root in linear_pair.roots
    report root
end
```

Expected x = 3, y = 2.

2 Periodic quadrature

```
workflow
quad periodic_area
    expr = sqrt(1-cos(2*x))
    var = x
    range = [0, 100*#pi + #pi/2]
end
report periodic_area.value
```

Expected 201*sqrt(2).

3 Compute functions

```
workflow
compute function_value
    expr = sin(#pi/2) + sqrt(36)
end
for row in function_value.result.results
    report row.value
end
```

Expected 7.

4 Differentiation

```
workflow
diff slope
    expr = x^3
    at x = 2
    wrt = x
end
report slope.result.requested_derivative
```

Expected first derivative 12 at x = 2.

5 Limits

```
workflow
limit sinc_limit
    expr = sin(x)/x
    at x = 0
end
report sinc_limit.value
```

Expected 1.

6 Interpolation

```
workflow
interpolate line_sample
    data = [[0,0], [1,2], [2,4]]
    method = linear
    at x = 1.5
end
for row in line_sample.result.results
    report row.value
end
```

Expected y = 3 at x = 1.5.

7 ODE initial value problem

```
workflow
ode exponential_decay
    y' = -y
    initial t = 0
    initial y = 1
    interval t = [0,1]
    output points = 3
end
point = at(exponential_decay.result.trajectory, 3)
report at(point.values, 2)
```

Expected y(1) = exp(-1).

8 Boundary value problem

```
workflow
bvp parabola
    y'' = -2
    interval x = [0,1]
    boundary left y = 0
    boundary right y = 0
    output step = 0.5
end
point = at(parabola.result.trajectory, 2)
report at(point.values, 2)
```

Expected y(0.5) = 0.25.

9 Poisson PDE

```
workflow
pde poisson_square
    type = poisson
    u_xx+u_yy = -2*pi^2*sin(pi*x)*sin(pi*y)
    domain x = [0,1]
    domain y = [0,1]
    boundary left u = 0
    boundary right u = 0
    boundary bottom u = 0
    boundary top u = 0
    grid x = 33
    grid y = 33
end
report poisson_square.status
```

Expected status `solution_computed`. The centre value u(0.5,0.5) is near 1 on this grid.

10 Eigenvalue problem

```
workflow
eigen fixed_ends
    type = sturm_liouville
    interval x = [0,1]
    p = 1
    q = 0
    weight = 1
    boundary left u = 0
    boundary right u = 0
    modes = 1
    grid x = 33
end
for mode in fixed_ends.result.eigenpairs
    report mode.eigenvalue
end
```

Expected first eigenvalue near pi^2.

11 Fredholm integral equation

```
workflow
integral fredholm_constant
    type = fredholm
    interval x = [0,1]
    forcing = 1-x/2
    kernel = x*s
    lambda = 1
    grid x = 65
end
point = at(fredholm_constant.result.trajectory, 1)
report at(point.values, 2)
```

Expected u(x) = 1.

12 Singularity analysis

```
workflow
singularities pole_scan
    expr = 1/(x-0.25)
    var = x
    range = [0,1]
end
for point in pole_scan.result.points
    report point.classification
end
```

Expected `infinite_singularity` for a pole at x = 0.25.